O'REILLY


图灵程序设计丛书

第3版



PostgreSQL 即学即用

PostgreSQL: Up and Running

涵盖PostgreSQL核心概念与功能特性

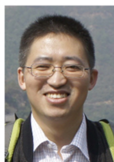
[美] 瑞金娜·奥贝 利奥·徐 著

丁奇鹏 译


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

译者介绍



丁奇鹏

2001年毕业于南京航空航天大学计算数学专业，先后在中兴和华为从事企业自有数据库的研发与技术管理等工作十余年，对关系型数据库的理论体系和内核实现均有深入研究，对于数据库技术在CT领域的应用拥有丰富的实践经验，对PostgreSQL和MySQL等开源数据库也有着浓厚兴趣。现就职于华为固定网络产品线，专注于大数据分析和机器学习领域的研究和落地。

数字版权声明


图灵程序设计丛书

PostgreSQL即学即用（第3版）

PostgreSQL: Up and Running
Third Edition

[美] 瑞金娜·奥贝　利奥·徐　著
丁奇鹏　译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（CIP）数据

PostgreSQL即学即用 : 第3版 / (美) 瑞金娜•奥贝
(Regina Obe) , (美) 利奥•徐 (Leo Hsu) 著 ; 丁奇鹏
译. -- 北京 : 人民邮电出版社, 2018.11
（图灵程序设计丛书）
ISBN 978-7-115-49966-0

Ⅰ. ①P… Ⅱ. ①瑞… ②利… ③丁… Ⅲ. ①关系数
据库系统 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2018)第246313号

内 容 提 要

本书将帮助你理解和使用 PostgreSQL 这一开源数据库系统。主要介绍 PostgreSQL 9.5、PostgreSQL 9.6 以及 PostgreSQL 10 的核心概念和功能特性，但也会涉及之前版本中一些独特的高级特性。你会发现 PostgreSQL 不只是个数据库系统，也是一个出色的应用平台。本书通过示例展示了如何实现在其他数据库中难以或不能完成的任务。

本书适合系统管理员、数据库管理员、后端开发人员，以及其他对 PostgreSQL 感兴趣的读者。

◆ 著　[美] 瑞金娜 · 奥贝　利奥 · 徐
译　丁奇鹏
责任编辑　岳新欣
责任印制　周昇亮

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800 × 1000　1/16
印张：16.25
字数：385千字　2018年 11 月第 1 版
印数：1 – 3 500册　2018年 11 月北京第 1 次印刷

著作权合同登记号　图字：01-2018-7656号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言

PostgreSQL 宣称自己是世界上最先进的开源数据库。我们非常赞同这种说法。

我们希望本书能帮助读者在 PostgreSQL 的核心概念与功能特性等方面打下坚实的基础，正是这些先进的概念与功能特性使得这款数据库如此杰出和与众不同。同时我们将使读者相信，PostgreSQL“最先进的开源数据库”这一称号是实至名归的。这款数据库体系庞大、功能先进，因此如果一本书试图将其强大特性介绍清楚，其篇幅一定不会少于 3500 页。事实上，大多数用户并不需要摸透它所有复杂深奥的高级功能，因此在这本不到 300 页的书中，我们希望能够帮助读者提纲挈领抓住要点，从而做到像本书书名所宣称的那样——即学即用。

本书在介绍每个功能点的同时都会附带其适用的上下文场景，这样读者就可以了解每个功能点的适用范围以及功能表现。我们假设读者已经具备了其他数据库的使用经验，这样我们就可以直奔主题，介绍 PostgreSQL 的关键功能点，而不必在预先普及数据库基础知识上浪费时间。我们在一些必要的地方附上了相关资源，以便于读者深入钻研感兴趣的功能点。这些资源的内容多种多样，比如官方手册的特定章节、网上有帮助的文章以及 PostgreSQL 大牛们的博客文章等，当然也包括我们自己的官网 Postgre Online Journal 上的一些文章，这个网站上有我们自己编写的很多 PostgreSQL 相关文章，也有一些研究 PostgreSQL 与其他应用之间互操作性的文章。

本书主要介绍 PostgreSQL 9.5、PostgreSQL 9.6 以及 PostgreSQL 10，但也会涉及之前版本中一些独特的高级特性。

本书读者

对于从其他数据库引擎迁移到 PostgreSQL 的读者来说，我们将在本书中列出其他数据库的高级特性在 PostgreSQL 中的实现方法。更重要的是，我们会重点介绍一些在 PostgreSQL 中可以实现而在其他数据库中很难或者不可能实现的高级功能。

本书不会教读者怎么写 SQL，相关学习资料有很多，因此这不是本书的重点。学习 SQL 就像学习下棋——几个小时就能了解基本规则，但要熟练掌握却需要终身持续学习。你会发现选择 PostgreSQL 是一个明智的决定，并将因此而受益终身。

如果你是对 PostgreSQL 非常熟悉的老用户或者是经验丰富的 DBA，那么本书中的大量内容你都会觉得很熟悉，但即便如此，你也一定可以学到新版 PostgreSQL 中引入的一些新特性，也很可能会了解到一些在老版本中已经提供但此前被你遗漏的功能点。好吧，如果你对于书中内容均已了解，那么本书对你来说依然有价值，因为它比官方的 PostgreSQL 手册要轻得多，最起码是便于携带了。

如果你还没接触过 PostgreSQL，那么本书将扮演你身边的 PostgreSQL“布道师”的角色。这位“布道师”将向你证明：多用那些很弱的数据库一天，你的系统就不得不多做一天功能上的妥协；多绑在商业数据库上一天，你就会被那些厂商掏走更多的钱。

最后，如果你的工作与数据库领域甚至是 IT 界毫无关系，又或者你刚刚幼儿园毕业，那么能否购买本书呢？答案依然是“可以”！因为封面上可爱的象鼩鼠图片就已经让本书物有所值了。

关于PostgreSQL的更多信息

PostgreSQL 有一套制作精良的在线文档，建议读者收藏它。这套文档有 HTML 和 PDF 两种格式，还有纸质印刷版。

其他可用的 PostgreSQL 资源如下。

- Planet PostgreSQL 是 PostgreSQL 技术博客文章的汇聚站点，其中包含从 PostgreSQL 核心开发人员到普通用户编写的各类文章，包括新特性用法介绍、原有特性的巧妙用法以及已发现但尚未正式修复的 bug 报告。
- PostgreSQL Wiki 提供对 PostgreSQL 各个方面的使用技巧说明，以及从其他数据库移植到 PostgreSQL 的方法。
- PostgreSQL Books 提供有关 PostgreSQL 的图书列表信息。
- PostGIS in Action Books 是我们已出版的关于 PostGIS 和 pgRouting 插件的图书的官方站点。PostGIS 是 PostgreSQL 上的空间数据管理插件，pgRouting 是基于 PostGIS 的一款网络路径规划插件，在导航出行类应用中经常要用到。

代码与输出格式

对于括号中的内容，我们一般会将左括号与之前的内容放置于同一行，右括号单独放置一行。这是经典的 C 语言排版风格，我们比较喜欢，因为它可以减少空行数。格式如下：

```
function (
        Welcome to PostgreSQL
);
```

为节省版面，我们还移除了命令行执行输出结果中无意义的空格，因此如果发现实际输出结果的格式与书中提供的不一致，请不要担心，这是正常的。

当一行中存在多个逗号时，如果每个元素的长度都比较短，我们会把逗号之后的空格去掉。比如：('a', 'b', 'c')。

PostgreSQL 的 SQL 解释器会将语句中的制表符、换行符和回车符当作空白处理。在我们提供的示例代码中，一般会使用空白而不是制表符作为缩进符。请确保使用的代码编辑器不会自动将制表符、换行符和回车符删除，或者把它们转换为空格以外的字符，否则会导致问题。

如果在执行示例代码时遇到了问题，请检查你复制过来的代码与我们提供的原始代码是否一致。

注意，有些示例适用于 Linux，而有些适用于 Windows。传统上，二者的路径分隔符是不同的，Linux 下是斜杠（`/`），而 Windows 下是反斜杠（`\`）。但请注意：在 PostgreSQL 中，即使是 Windows 环境下，也一定要使用 Linux 的 `/` 作为路径分隔符，而不是 Windows 传统的 `\`。你会看到示例代码中有类似于 `/postgresql_book/somefile.csv` 这样的路径，这指的是 Linux 服务器根目录下的路径。如果使用的是 Windows 环境，那么需要加上驱动器符，因此路径要改为：`C:/postgresql_book/somefile.csv`。

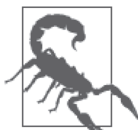
排版约定

本书将使用如下排版约定。

- **黑体**
  表示新术语。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽粗体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 斜体等宽斜体（*`constant width italic`*）
  表示应替换成用户提供的值或由上下文决定的值。



该图标表示提示或建议。



该图标表示警告或警示。

使用代码示例

代码和数据示例可以从 http://www.postgresonline.com/downloads/postgresql_book_3e.zip 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*PostgreSQL*: *Up and Running*, *Third Edition* by Regina Obe and Leo Hsu (O'Reilly). Copyright 2018 Regina Obe and Leo Hsu, 978-1-491-96341-8."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O'Reilly Safari

Safari（原来叫 Safari Books Online）是一个会员制的培训和参考平台，面向企业、政府、教育机构和个人。

会员可以访问几千种图书、培训视频、学习路径、互动式教程和精选播放列表，提供这些资源的出版商超过 250 家，包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology，等等。

要获得更多信息，请访问 http://oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要提交勘误，请访问本书的勘误页面（https://www.oreilly.com/catalog/errata.csp?isbn=0636920052715）。[1]

注 1：本书中文版勘误请到 http://www.ituring.com.cn/book/2460 查看和提交。——编者注

本书的配套网站地址是：https://www.oreilly.com/catalog/errata.csp?isbn=0636920052715

要联系本书作者，请发送邮件到 lr@pcorp.us。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多有关 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

电子书

扫描如下二维码，即可购买本书电子版。



第 1 章

基础知识

PostgreSQL 是一款极其强大的数据库，它的很多特性可能是你前所未见的。它的部分特性在其他知名数据库中也有，但名称可能不同。在深入钻研官方手册之前，你需要了解一些核心概念，本章将为你介绍这些概念，期间也会涉及其他数据库中的相关概念和术语。

本章将首先介绍如何下载和安装 PostgreSQL，然后会介绍一些必备的管理工具和 PostgreSQL 术语。本书写作之时，PostgreSQL 10 已发布，我们将重点介绍该版本的一些新特性。本章末尾会提供一些帮助资源，当你需要额外的帮助或者报告 bug 时会用得到。

1.1 为什么应该选择PostgreSQL

PostgreSQL 是一款企业级关系型数据库管理系统，即使与 Oracle、Microsoft SQL Server、IBM DB2 等业界最好的商用数据库相比也毫不逊色。PostgreSQL 之所以如此特别，是因为它不仅仅是一个数据库，还是一个功能强大的应用开发平台。

PostgreSQL 的速度很快。大量的评测数据已经表明：与其商用以及开源竞争对手相比，PostgreSQL 的速度要么远远胜出，要么旗鼓相当。

PostgreSQL 支持用多种编程语言编写存储过程和函数。除了系统自带的 C、SQL 和 PL/pgSQL 编程语言外，还可以通过安装扩展包来支持 PL/Perl、PL/Python、PL/V8（又称为 PL/JavaScript）、PL/Ruby 以及 PL/R 等。 这种支持多语言的能力可以让开发人员根据待解决问题的特点来选择最合适的语言。比如可以使用 R 语言来解决统计和图形领域的问题，通过 Python 来调用 Web 服务，通过使用 SciPy 库来进行科学计算，通过 PL/V8 来进行数据验证、字符串处理和 JSON 数据处理，等等。PostgreSQL 不但支持种类繁多的开发语言，使用过程也很简单：先找到你需要的函数，看下它是用什么语言编写的，在 PostgreSQL 中安装好支持该语言的扩展包，然后把代码复制过来就可以执行了。真的不能更简单。

大多数数据库会限制用户只能使用预定义的数据类型，比如整型、字符串、文本型、布尔型等。PostgreSQL 在数据类型的支持方面有两个优势：不但支持比绝大多数数据库更丰富的原生数据类型，而且还允许用户按需自定义数据类型。如果用户需要更复杂的数字类型，那么可以定义包含两个浮点类型（float）的复合类型；如果需要定义一个三角形，那么可以先定义一种“坐标”类型，然后再定义一种包含三个坐标的“三角形”类型即可；如果你对十二进制感兴趣，那么可以定义你自己的十二进制类型。值得注意的是，要想自定义类型完全发挥出其威力，需要有相应的运算符和函数来识别并配合它。因此，如果你自定义了一种特殊的数值类型，千万不要忘了为它重定义配套的数学运算符。是的，你没看错，PostgreSQL 允许用户重定义基础运算符（+、-、/、*）的实现逻辑。另外，用户自定义一种数据类型后，PostgreSQL 会自动定义出一种基于该类型的数组类型。因此，如果你定义了一种复合数据类型，那么该复合数据类型的数组类型自动就有了，你无须做额外的定义工作。

PostgreSQL 会为每一张用户表自动创建一个数据类型定义。比如我们创建了一张名为 `dogs` 的表，包含 breed（品种）、cuteness（可爱程度）、barkiness（爱叫的程度）等字段，那么 PostgreSQL 会自动在后台创建一个同名的 `dogs` 数据类型。这一特性将关系型数据库领域的“表”概念与面向对象领域的“对象”概念紧密地联系到了一起，用户可以像处理对象实例一样去处理记录。比如你可以创建一个函数来每次处理一个或者一批对象实例。PostgreSQL 的很多第三方扩展包就利用该自定义数据类型能力来优化性能，或者通过添加支持某个领域专用的特殊 SQL 语法来让业务代码更简洁和易于维护，或者实现一些在别的数据库中完全不可能实现的功能。

我们建议用户不要把数据库仅仅当成一个数据容器，像 PostgreSQL 这样的数据库其实是一个成熟而完整的应用开发平台。你会发现：强大的数据库在手，其他一切都是过眼云烟。一旦你成了 SQL 高手，别人用其他工具需要几小时才能完成的工作，你在数据库里只需要几秒钟。不管是对比编码时间还是对比实际数据处理时间，效果都是如此。

近年来，我们看到很多 NoSQL 数据库异军突起，一时风头无两（我们认为这里面有很大的炒作成分）。尽管 PostgreSQL 总体上看是一个关系型数据库，但它其实也具备强大的处理非关系型数据的能力。比如 ltree 这个插件可以处理图数据，但我们几乎已经想不起它到底是何时被加入到 PostgreSQL 中的；又比如 hstore 插件可以实现键值存储；还有 JSON 和 JSONB 类型可以提供类似 MongoDB 的文档操作能力。从很多方面来看，PostgreSQL 甚至在“NoSQL”这个词出现之前就已经提供了那些所谓的 NoSQL 特性。

如果从 Postgres95 正式改名为 PostgreSQL 开始算起，PostgreSQL 已经诞生二十年了，但事实上它的历史可向前一直追溯到 1986 年。[1]PostgreSQL 支持当前所有主流的操作系统，包括 Linux、Unix、Windows 以及 Mac。目前每年会发布一个大版本，包含性能提升以及那些不断超越关系型数据库功能极限的新功能。

最后值得一提的是，PostgreSQL 不但是开源的，而且开源得非常彻底，它的许可策略非

注 1：从 1986 年开始，Stonebraker 教授发表了一系列论文，引入对象关系理念，探讨了新的数据库的结构设计和扩展设计。1988 年，他发表了 Postgres 的第一个原型设计，1989 年 6 月发布了版本 1。因此 1986 年可视为 PostgreSQL 发展史的元年。——译者注

常宽松。现在 PostgreSQL 社区由一群无私奉献的的开发者和用户构成，他们并不把赚钱视为人生终极目标。如果需要新特性，你可以自行贡献代码或者提出诉求。如果你试图修改 PostgreSQL 代码以为己所用，也不会有人起诉你。是成千上万用户的参与和贡献使得 PostgreSQL 变成了它今日的模样。

到最后你会想：我为什么还要使用别家的数据库？ PostgreSQL 已经提供了我所需要的一切功能，而且还是免费的！你不再需要去阅读那些商业数据库附带的密密麻麻的授权条款，来了解在一个八核虚拟机上支持 X 个并发连接所需要的费用是多少，也不需要了解每次升级后要再为许可证加多少钱。

1.2 不适用PostgreSQL的场景

在为 PostgreSQL 做了这么多“鼓吹”之后，我们也应介绍一下它不适用于哪些场景。

在不安装任何扩展包的情况下，PostgreSQL 需占用 100MB 以上的磁盘空间，可以看出它的个头还是比较大的。因此，在一些存储空间极为有限的小型设备上使用 PostgreSQL 是不合适的，把 PostgreSQL 当成简单的缓存区来用也是不合适的，此时应选用一些更轻量级的数据库。

作为一款企业级数据库产品，PostgreSQL 对于安全是极其重视的。因此，如果你在开发一个把安全管理放到应用层去做的轻量级应用，那么 PostgreSQL 完善的安全机制反倒会成为负担，因为它的角色和权限管理非常复杂，会带来不必要的管理复杂度和性能损耗。此时可以考虑选用 SQLite 这样的单用户数据库，或者选用 FireBird 这样既能以客户端 / 服务器模式运行也能以嵌入式单用户模式运行的数据库。

通过上述介绍可以看到，一般来说需要将 PostgreSQL 与别的数据库搭配使用，使它们各展所长。一种常见的组合是把 Redis 或者 Memcache 当成 PostgreSQL 的查询缓存来用；另一种组合是用 PostgreSQL 做主数据库，用 SQLite 存储离线数据来做离线查询。

令人遗憾的一个事实是，很多共享主机服务（多个用户共享同一个操作系统实例）供应商并不支持预安装 PostgreSQL，或者只支持安装一个很陈旧的版本。它们更喜欢预装较弱的 MySQL。对于一个 Web 设计人员来说，用什么数据库并不是首要问题，此时 MySQL 可能能够满足要求。但当用户的 SQL 技能不断提升，不再满足于写一写单表 select 或者简单的 join 查询时，MySQL 的缺点就会暴露无遗。自本书第一版出版以来，虚拟化技术的进步使得商业化的云主机服务得到了长足的发展，因此拥有自己的独立云主机不再是一件很奢侈的事情。当用户拥有自己的独立云主机时，就可以自由选择在上面安装什么软件。随着 PaaS（平台即服务）、DBaaS（数据库即服务）等云计算模式的流行，PostgreSQL 的发展前景向好。绝大多数的云服务厂商都提供 PostgreSQL 服务，其中比较著名的有 Heroku、Engine Yard、RedHat OpenShift、Amazon RDS for PostgreSQL、Google Cloud SQL for PostgreSQL、Amazon Aurora for PostgreSQL 以及 Microsoft Azure for PostgreSQL。

1.3 如何获得PostgreSQL

若干年前，你只能通过手动编译源码的方式来安装 PostgreSQL。还好那种痛苦的时代已经

一去不复返了。当然，现在依然可以通过编译源码来安装，但大多数用户会使用制作好的安装包来安装，只需敲击几下键盘和鼠标就可以了。

如果你是首次安装 PostgreSQL，那么应该选择适用于你的操作系统平台的最新稳定版发行包。PostgreSQL 官方站点的核心发布页面上维护了一个列表，记录了适用于各操作系统的二进制包的下载地址。在附录 A 中，你会看到安装指导和一些定制版本的下载链接地址。

1.4 管理工具

PostgreSQL 常用的管理工具有四种：psql、pgAdmin、phpPgAdmin 和 Adminer。PostgreSQL 的核心开发团队维护着前三种，因此它们一般会随着 PostgreSQL 的版本发布而同步更新。Adminer 并非 PostgreSQL 的专用管理工具，它支持管理多种类型的关系型数据库，包括 SQLite、MySQL、SQL Server 和 Oracle。除了刚刚提到的这四种以外，还有大量优秀的管理工具，开源的和商业的都有。

1.4.1 psql

psql 是一种用于执行查询的命令行工具，每个 PostgreSQL 发行版中都自带 psql（参见附录 B.4 节）。它有一些独特的功能，比如导入和导出基于分隔符（逗号或者制表符等）格式的平面数据文件，以及生成简易的 HTML 格式报表等。psql 是 PostgreSQL 从诞生之初就一直附带的命令行工具，它是很多高级用户日常操作工具的不二之选，非常适用于只有控制台字符界面而无图形用户界面的使用场景。另外，在通过 shell 脚本执行数据库操作时，psql 也是必备工具。不过新用户一般更喜欢使用图形界面工具，而且也无法理解为什么“老”一代人会对命令行方式那么执着。

1.4.2 pgAdmin

pgAdmin 是一款流行的免费的 PostgreSQL 图形界面管理工具。如果你的 PostgreSQL 安装包里没有附带此工具，请从其官网单独下载安装。pgAdmin 可在 PostgreSQL 支持的任意一种操作系统平台上运行。

即使你的数据库安装在只有控制台字符界面的 Linux 服务器上，只要你在本地工作站上安装了 pgAdmin，也可以用这种强大的图形化工具对其进行管理。

pgAdmin 近期已经发布了它的第四个大版本，称为 pgAdmin4。该版本对之前的 pgAdmin3 进行了彻底的重写，使用 Python 实现了“一套代码两种模式运行”的效果，一种模式是作为桌面应用运行，另一种是在浏览器中运行。pgAdmin4 当前的版本是 1.5。pgAdmin4 的首个版本是与 PostgreSQL 9.6 同时发布的，并被若干 PostgreSQL 发行版作为自带软件一起打包发布。如前所述，pgAdmin4 既可以作为桌面应用运行，也可以在浏览器中运行。

图 1-1 是 pgAdmin4 的界面示意图。

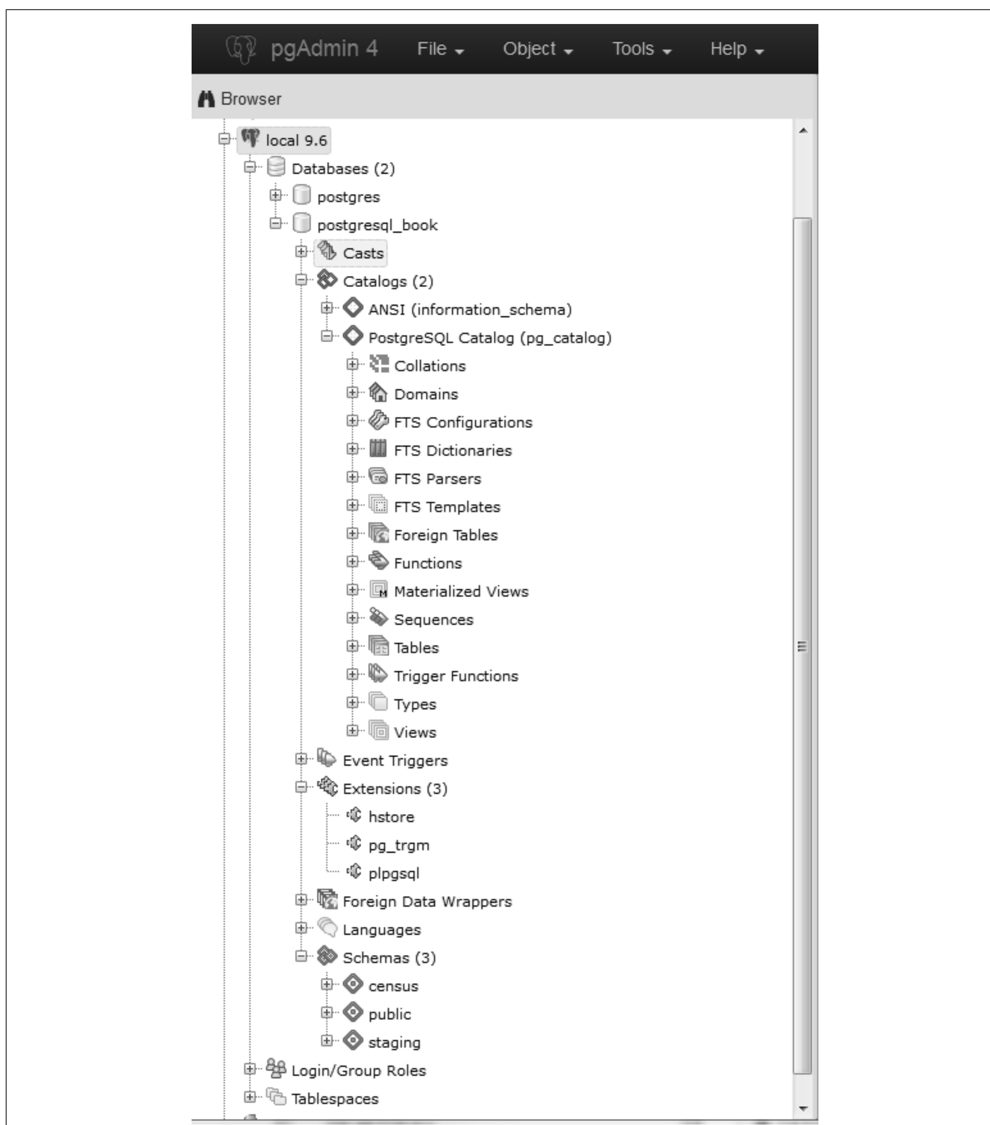


图 1-1：pgAdmin4 的树状视图

如果你对 PostgreSQL 还不太熟悉，那么 pgAdmin 毫无疑问是你开始 PostgreSQL 学习之旅的最佳入口。只需在主界面上摸索一下，你就可以对 PostgreSQL 的丰富功能一览无遗。如果你正打算逃离 Microsoft SQL Server 阵营，并且习惯于 SQL Server 的 Management Studio，那么很快就能适应 pgAdmin。

相比 pgAdmin3，pgAdmin4 还有一些短板，但它正在快速补齐并在很多方面都超过了 pgAdmin3。即便如此，如果你是 PgAdmin3 的长期用户并且短期内无法切换到 pgAdmin4，那么你可以继续使用 BigSQL 公司提供的 pgAdmin3 LTS（长期支持）版，在对 pgAdmin4

进行完善测试后再切换过去。请务必牢记，pgAdmin4 才是 pgAdmin 未来的主力版本，pgAdmin3 只会维持现状，不会再有什么发展。

1.4.3 phpPgAdmin

phpPgAdmin 是一种免费的基于 Web 页面的管理工具，其界面如图 1-2 所示。它是从流行的 MySQL 管理工具 phpMyAdmin 移植而来的，二者的差别主要在于 phpPgAdmin 新增了对 PostgreSQL 的 schema、过程式语言、类型转换器、运算符等对象的管理功能。如果你对 phpMyAdmin 很熟悉，会发现 phpPgAdmin 的界面风格与其完全类似。



图 1-2：phpPgAdmin

1.4.4 Adminer

如果你正在寻找一款除了能够管理 PostgreSQL，还能管理别的数据库的整合型工具，那么 Adminer 将是你合适的选择。Adminer 是一款轻量级的开源 PHP 应用程序，可以在同一套图形界面上管理 PostgreSQL、MySQL、SQLite、SQL Server 以及 Oracle 等多种数据库。

Adminer 有一种独特的功能让我们印象深刻：它能够以图形化方式展示数据库中的对象，并将外键约束关系以连接线的方式展示出来。另外，整个 Adminer 程序的本体仅包含一个 PHP 文件，非常简洁，这可以大大减少你安装部署时的麻烦。

图 1-3 中，左侧是登录屏幕的截图，右侧是表间关系图形化后呈现的效果。很多用户会因为登录屏幕上没有填写端口号的地方而感到困惑。如果 PostgreSQL 使用标准的 5432 侦听端口，那么登录时不填也没问题；但如果不是，就需要在服务器名称后面加上端口号，注意用冒号分隔主机名和端口号，如图 1-3 所示。

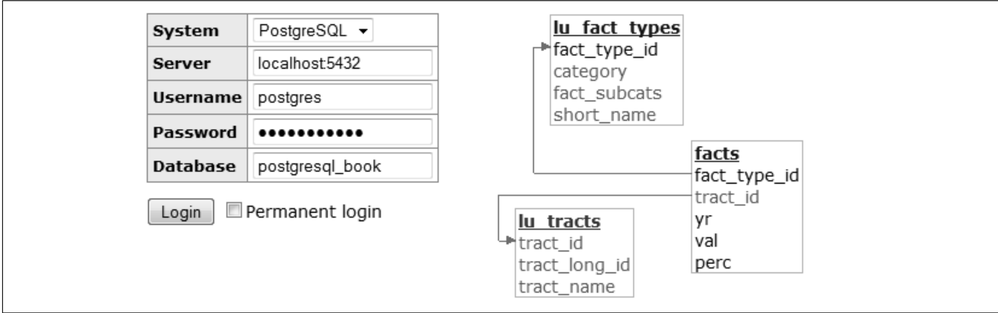


图 1-3：Adminer

对于简单的查询和修改操作来说，Adminer 的功能是足够的。但为了支持多种数据库，Adminer 的功能体系已经被裁剪成了各数据库均支持的最小公共集合，因此你无法实现 PostgreSQL 所特有的一些操作，比如创建新用户、授予权限、查询当前权限列表等。Adminer 为了与它所支持的各家数据库在概念上保持兼容和通用而将每个 schema 当作一个 database，这使得以图形化展示表与表之间外键关系这一功能受到了极大影响，如果两个不同 schema 的表之间存在外键关联关系，那么在 Adminer 的界面上是无法展示出来的。如果你是 DBA，那么建议使用 pgAdmin，当然也可以安装一套 Adminer 以备不时之需。

1.5　PostgreSQL数据库对象

假设你现在已经安装好了 PostgreSQL，请启动并连接好 pgAdmin，然后点开左侧的目录树，此时展现在你面前的是一堆令人眼花缭乱的数据库对象，有些你可能很熟悉，有些则可能闻所未闻。PostgreSQL 对象类型的数量超过了绝大多数关系型数据库（这还是在未安装任何扩展包的情况下）。这些对象中，有许多你可能永远都不会用到，但如果你发现业务上需要实现一种新的对象类型，那么一般来说你要实现的东西在那一堆眼花缭乱的对象中已经有前人实现过了，所以只需要正确选用即可。本书不会介绍 PostgreSQL 以标准方式安装完毕后所提供的所有对象类型，因为 PostgreSQL 引入新特性的速度惊人，任何一本书都不可能全面覆盖所有对象类型。因此我们仅讨论你有必要了解的那些对象类型。

database[2]

　每个 PostgreSQL 服务可以包含多个独立的 database。

schema[3]

　ANSI SQL 标准中对 schema 有着明确的定义，database 的下一层逻辑结构就是 schema。

注 2：database 一词含义宽泛，既可表示广义的数据库系统，又可以表示某些特定数据库系统中的某一级数据存储单位，如表述不当极易给读者造成混淆。因此本书中会区别使用，表示广义的数据库系统时，用中文“数据库”；表示狭义的数据存储单位时，用英文“database”。——译者注

注 3：数据库业界对于 schema 有多种译法：纲要、模式、方案，等等。但各种译法都不能准确直观地表达出其原本的含义，即位于一个独立命名空间内的一组相关数据库对象的集合，因此前述译法从来没有一种成为主流。一般业界人员都直接使用英文 schema。考虑到这个情况，为防止初级用户理解困难，我们也按照业界习惯直接使用英文原名。——译者注

如果把 database 比作一个国家，那么 schema 就是一些独立的州（或者是省、府、辖区等，具体取决于各国的实际情况）。大多数对象是隶属于某个 schema 的，然后 schema 又隶属于某个 database。在创建一个新的 database 时，PostgreSQL 会自动为其创建一个名为 `public` 的 schema。如果未设置 `search_path` 变量（后续会介绍该变量的含义），那么 PostgreSQL 会将你创建的所有对象默认放入 `public` schema 中。如果表的数量较少，这是没问题的，但如果你有几千张表，那么我们还是建议你将它们分门别类放入不同的 schema 中。

表

任何一个数据库中，表都是最核心的对象类型。在 PostgreSQL 中，表首先属于某个 schema，而 schema 又属于某个 database，这样就构成了一种三级存储结构。

PostgreSQL 的表支持两种很强大的功能。第一种是表继承，即一张表可以有父表和子表。这种层次化的结构可以极大地简化数据库设计，还可以为你省掉大量的重复查询代码。第二种是创建一张表的同时，系统会自动为此表创建一种对应的自定义数据类型。

视图

大多数关系型数据库都支持视图。视图是基于表的一种抽象，通过它可以实现一次性查询多张表，也可以实现通过复杂运算来构造出虚拟字段。视图一般是只读的，但 PostgreSQL 支持对视图数据进行修改，前提是该视图基于单张实体表构建。如果需要修改基于多张表关联而来的视图，可以针对视图编写触发器。9.3 版还引入了对物化视图的支持，该机制通过对视图数据进行缓存来实现对常用查询的加速，缺点是查到的数据可能不是最新的。更多细节请参见 7.1.3 节。

扩展包

开发人员可以通过该机制将一组相关的函数、数据类型、数据类型转换器、用户自定义索引、表以及属性变量等对象打包成一个功能扩展包，该扩展包可以整体安装和删除。扩展包在概念上与 Oracle 的 package 类似，从 PostgreSQL 9.1 版本之后一般推荐使用该机制来为数据库提供功能扩展。扩展包的具体安装步骤，请参考开发手册。一般来说，需要先将扩展包的二进制安装包和脚本复制到 PostgreSQL 安装目录下，然后运行一系列脚本，再在需要其功能的 database 中单独安装该扩展包。注意：仅需在需要该扩展包功能的 database 中安装，不必为当前数据库系统中的每个 database 都安装。比如需要对某个 database 中的数据进行高级文本搜索，那么单独在该 database 中安装 `fuzzystrmatch` 扩展包即可。

安装扩展包时可以指定该包中所含有的成员对象安装到哪个 schema，若不指定则默认会安装到 `public` schema 中。我们不建议采用默认设置，因为这会导致 `public` schema 变得庞大复杂且难以管理，尤其是如果你将自己的数据库对象也都存入 `public` schema 中，那么情况会变得更糟糕。我们建议你创建一个独立的 schema 用于存放所有扩展包的对象，甚至为规模较大的扩展包单独创建一个 schema。为避免出现找不到新增扩展包对象的问题，请将这些新增的 schema 名称加入 `search_path` 变量中，这样就可以直接使用扩展包的功能而无须关注它到底安装到了哪个 schema 中。也有一些扩展包明确要求必须安装到某个 schema 下（特别是过程式语言扩展包），这种情况下你就不能自行指定了。有很多语言扩展包，比如 `plv8`，就要求必须安装到 `pg_catalog` schema 中。

多个扩展包之间可能存在依赖关系。在 PostgreSQL 9.6 之前，你需要了解这个依赖关系并把被依赖包先装好，但从 9.6 版开始，只需在安装时加上 cascade 关键字，PostgreSQL 就会自动安装当前扩展包所依赖的扩展包。例如：

```
CREATE EXTENSION postgis_tiger_geocoder CASCADE;
```

这条命令会先寻找并安装被依赖的 postgis 和 fuzzystrmatch 这两个扩展包，当然，如果原本就有了，就不需要了。

函数

用户可以编写自定义函数来对数据进行新增、修改、删除和复杂计算等操作，可以使用 PostgreSQL 所支持的各种过程式语言来编码。PostgreSQL 装好以后自身就包含了数以千计的系统函数，都在默认的 postgres 库中。函数支持返回以下数据类型：标量值（也就是单个值）、数组、单条记录以及记录集。其他数据库将对数据进行增删改操作的函数称为“存储过程”，把不进行增删改的函数叫作“函数”，但 PostgreSQL 中并不区分，统一把二者称为“函数”。

内置编程语言

函数是以过程式语言编写的。PostgreSQL 默认支持三种内置编程语言：SQL、PL/pgSQL 以及 C 语言。可以通过 CREATE EXTENSION 或者 CREATE PRODCEDURAL LANGUAGE 命令来添加其他语言。目前较常用的语言是 PL/Python、PL/V8（即 JavaScript）以及 PL/R。我们将在第 8 章中展示大量的相关示例。

运算符

运算符本质上是以简单符号形式呈现的函数别名，例如 =、&& 等。PostgreSQL 支持自定义运算符。如果用户定义了自己的数据类型，那么一般来说需要再自定义一些运算符来与之配合工作。比如你定义了一个复数类型，那么你很有可能需要自定义 +、-、*、/ 这几个运算符来对复数进行运算。

外部表和外部数据封装器

外部表是一些虚拟表，通过它们可以直接在本地数据库中访问来自外部数据源的数据。只要数据映射关系配置正确，外部表的用法就与普通表没有任何区别。外部表支持映射到以下类型的数据源：CSV 文件、另一个服务器上的 PostgreSQL 表、SQL Server 或 Oracle 这些异构数据库中的表、Redis 这样的 NoSQL 数据库，甚至像 Twitter 或 Salesforce 这样的 Web 服务。

外部表映射关系的建立是通过配置外部数据封装器（foreign data wrapper，FDW）实现的。FDW 是 PostgreSQL 和外部数据源之间的一架“魔法桥”，可实现两边数据的互联互通。其内部实现机制遵循 SQL 标准中的 MED（Management of External Data）规范，更多细节请参考维基百科上关于 MED 的描述。

许多无私的开发者已经为当下大部分流行的数据源开发了 FDW 并已免费共享出来。你也可以通过创建自己的 FDW 来练习。（我们建议你一旦成功了也公布出来，这样整个社区都可以分享你的劳动成果。）FDW 是通过扩展包机制实现的，安装好以后在 pgAdmin 界面上名为 Foreign Data Wrapper 的目录节点下能看到它。

触发器和触发器函数

绝大多数企业级数据库都支持触发器机制，该机制可以侦测到数据修改事件的发生。在 PostgreSQL 中，当一个触发器被触发后，系统会自动调用用户定义好的触发器函数。触发器的触发时机是可设置的，可以是语句级触发或者记录级触发，也可以是修改前触发或修改后触发。

在 pgAdmin 中，如果希望了解哪些表上挂载了触发器，只需在对象树上层层点击，一直打开到表这一级，然后可以看到下面有个 trigger 子栏目，里面就是该表的所有触发器。

定义触发器时需要定义对应的触发器函数，这类函数与前面介绍过的普通函数有所不同，主要差异在于触发器函数可以通过系统内置变量来同时访问到修改前和修改后的数据，这样就可以实现对于非法的数据修改行为的识别和拦截。因此触发器函数一般会用于编写复杂校验逻辑，这类复杂逻辑通过 check 约束是无法实现的。

PostgreSQL 的触发器技术正在快速的演进之中。9.0 版引入了对 `WITH` 子句的支持，通过它可以实现带条件的记录级触发，即只有当某条记录符合指定的 `WHEN` 条件时，触发器才会被调用。9.0 版还引入了 `UPDATE OF` 子句，通过它可以实现精确到字段级的触发条件设置。仅当指定的字段内容被更改时才会激活触发器。9.1 版支持了针对视图的触发器。9.3 版支持了针对 DDL 的触发器。目前支持触发器的 DDL 命令列表请参见官方手册中“触发器触发时机一览表”。pgAdmin 中会把 DDL 触发器列在 event trigger 这一栏下。最后值得一提的是，从 9.4 版开始，针对外部表的触发器也获得了支持。

catalog[4]

catalog 是系统级的 schema，用于存储系统函数和系统元数据。每个 database 创建好以后默认都会含有两个 catalog：一个名为 `pg_catalog`，用于存储 PostgreSQL 系统自带的函数、表、系统视图、数据类型转换器以及数据类型定义等元数据；另一个是 `information_schema`，用于存储 ANSI 标准中所要求提供的元数据查询视图，这些视图遵从 ANSI SQL 标准的要求，以指定的格式向外界提供 PostgreSQL 元数据信息。

一直以来，PostgreSQL 数据库的发展都严格地遵循着其“自由与开放”的核心理念。如果你足够了解这款数据库，会发现它几乎是一种可以“自我生长”的数据库。比如，它所有的核心设置都保存在系统表中，用户可以不受限地查看和修改这些数据，这为 PostgreSQL 提供了远超任何一种商业数据库的巨大灵活性（不过从另一个角度看，将这种灵活性称为“可破坏性”也未尝不可）。只要仔细地研究一下 `pg_catalog`，你就可以了解到 PostgreSQL 这样一个庞大的系统是如何基于各种部件构建起来的。如果你有超级用户权限，那么可以直接修改 `pg_catalog` 的内容（当然，如果改得不对，那你的行为就跟搞破坏没什么两样了）。

`Information_schema` catalog 在 MySQL 和 SQL Server 中也有。PostgreSQL 的 `Information_schema` 中最常用的视图一般有以下几个：`columns` 视图，列出了数据库中的所有字段；`tables` 视图，列出了数据库中的所有表（包括视图）；`views` 视图，列出了所有视图以及用于创建该视图的原始 SQL。

注 4：catalog 的译法与 schema 存在相同的问题，翻译为“目录”后并不能让读者准确地理解其原意，反而容易造成混淆，因此还是沿用英文原名。——译者注

类型

类型是数据类型的简称。每种数据库产品和每种编程语言都会支持一系列的数据类型，比如整型、字符型、数组、二进制大对象（blob）等。除前述常见类型外，PostgreSQL 还支持复合数据类型，这种类型可以是多种数据类型的一个组合，比如复数、极坐标、向量、张量等都是复合数据类型。

PostgreSQL 会自动为用户自己创建的表定义一个同名的复合数据类型。这样就可以把表记录当作对象实例来处理。当用户需要在函数中遍历表记录时，该特性特别有用。注意：在 pgAdmin 的界面上你看不到这些在创建表时自动生成的自定义类型，但请放心，这并不代表它们不存在。

全文检索

全文检索（full text search，FTS）是一种基于自然语言的搜索机制。这种搜索机制有一些"智能"成分。与正则表达式搜索不同，全文检索能够基于语义来进行匹配查找，而不仅仅是纯粹的语法匹配。例如，用户需要在一段长文本中搜索 running 这个词，那么命中的结果可能包含 run、running、jog、sprint、dash 等词。全文检索功能依赖于 FTS 配置库、FTS 词典、FTS 解析器这三个部件。有了它们，PostgreSQL 原生的 FTS 功能即可正常使用。一般场景下的全文检索靠这三个原生部件已经足够，但在涉及药理学、有组织犯罪学等专业场景下，搜索目标文本中会包括该领域专有词汇和特殊语法规则，此时需要用专门的 FTS 部件来替换原生 FTS 部件。我们会在 5.8 节探讨 FTS 功能。

数据类型转换器

数据类型转换器可以将一种数据类型转换为另一种，其底层通过调用转换函数来实现真正的转换逻辑。PostgreSQL 支持用户自定义转换器或者重载、加强默认的转换器。例如，如果你需要把邮政编码（美国的邮政编码是一个 5 位的整数）从 `integer` 转换为 `character`，那么可以自定义一个支持"数字不足 5 位则前面自动补 0"规则的转换器。

转换器可以被隐式调用也可以被显式调用。隐式转换是系统自动执行的，一般来说，将一种特定数据类型转为更通用的数据类型（比如数字转换为字符串）时就会发生隐式类型转换。如果进行隐式转换时系统找不到合适的转换器，你就必须显式执行转换动作。

序列号生成器

序列号生成器用于实现 `serial` 数据类型值的自动递增分配。在创建 `serial` 字段时，PostgreSQL 会自动为其创建一个相应的序列号生成器，但用户也可以很方便地更改其初始值、步长和下一个值。因为序列号生成器是独立对象，所以多个表可以共享同一个序列号生成器。基于该机制，用户可以实现跨越多个表的唯一键。SQL Server 和 Oracle 也都支持序列号生成器，但必须手动创建。

规则

规则的功能是在一个 SQL 执行前对其进行改写。本书中不会讨论有关规则的内容，因为这一技术已经过时，通过触发器能实现相同的功能。

PostgreSQL 允许用户对前述每一种对象进行参数配置。这些参数可以在服务级、库级、函数级等不同层级生效。你将来很可能会看到一个很炫的词叫 GUC，意思是"大一统配置"（grand unified configuration），它实际上指的就是 PostgreSQL 中的那些配置参数。

1.6 最新版本的PostgreSQL中引入的新特性

PostgreSQL 在每年的 9 月份会发布一个大版本。每个新版本都会带来稳定性、安全性、性能等方面的提升，以及一些前沿的新特性。而且版本升级过程也变得越来越简单。那么显而易见，请尽量把你的数据库及时升级到最新的稳定版。关于每个版本引入的关键特性列表，请参见官方提供的“PostgreSQL 各版本功能特性一览表”。

1.6.1 为什么要升级

如果你正在使用 PostgreSQL 9.1 或者更早的版本，请立即升级！因为 9.1 版在 2016 年 9 月已进入生命周期终结（end of life，EOL）状态。请参考 PostgreSQL 官方的发行版支持策略以获取更多关于 PostgreSQL EOL 政策的细节。请务必不要使用已过了 EOL 期限的版本，因为开发组不会再为其提供新的安全更新和功能补丁。一旦这种老版本出了问题，你只能花钱去请 PostgreSQL 专家级顾问来解决故障或寻找临时解决方案，这种服务一般都是很昂贵的，而且你不一定能找得到这种专家。

不管当前使用的是哪个大版本，你都应该尽快跟进小版本号的更新。比如从 9.1.17 升级到 9.1.21，只需要替换二进制文件并重启一下即可。小版本仅修改 bug 而不会涉及功能变化，因此这种升级是很安全的，也会为你降低出问题的概率。

1.6.2 PostgreSQL 10中引入的新特性

PostgreSQL 10 是目前最新的稳定版，于 2017 年 10 月发布。从 PostgreSQL 10 开始，PostgreSQL 会以一种新的方式升级其版本号。在之前的版本中，发布大版本时变化的是第二位小版本号，比如从 PostgreSQL 9.5 到 PostgreSQL 9.6，即使增加了一些比较大的新功能，也只有小版本号发生变化。但从 PostgreSQL 10 开始，每个变化较大的版本都会在主版本号上加一。因此 PostgreSQL 10 的下一个大版本是 PostgreSQL 11。这样就与 SQLite、SQL Server、Oracle 等数据库的版本号策略保持了一致。

以下是 PostgreSQL 10 中引入的关键新特性。

提升了查询的并行度

对于并行查询启用了新的优化策略，包括并行位图堆扫描、并行索引扫描等。这些增强将使得更多查询语句能被并行执行。请参考 9.4 节以了解更多信息。

逻辑复制

此前版本的 PostgreSQL 中已经支持流复制特性。通过流复制可以实现整个 PostgreSQL 服务实例的复制，但该机制有一些固有的缺点：从节点是只读的，只能用于数据查询，不能对其数据进行修改；从节点上也不能创建自己独有的表。逻辑复制解决了这两个问题。通过逻辑复制可以实现仅复制单张表或者单个 database（不用复制整个服务实例的所有数据）。既然不需要复制整个数据库服务，那么自然从库上就可以有自己的表和数据，这部分数据是不包含在复制体系中的，因此主从库上允许不一样。

针对 JSON 和 JSONB 类型的全文检索

此前的版本中，`to_tsvector` 函数仅能为文本类型的字段生成全文检索向量。现在它已

支持处理 JSON 和 JSONB 类型，处理过程中会忽略其中 key 的部分，而仅包含 value 的部分。同时 `ts_headline` 函数也专为 JSON 和 JSONB 类型做了适配，它可以对 JSON 内容中的关键字进行加亮标记。详情请参考 5.8.7 节。

支持 ANSI 标准中的 `XMLTABLE` 特性

`XMLTABLE` 特性可以将 XML 文本内容以一种更为简单的方式映射为普通二维表记录。该特性在 Oracle 和 IBM DB2 中已支持。详情请参见示例 5-41。

FDW 聚合下推

FDW API 可以将 `COUNT(*)` 或者 `SUM(*)` 这种聚合操作推送到远端节点执行。`postgres_fdw` 插件从该特性中受益最大。此前的版本中，`postgres_fdw` 插件在执行聚合操作时，需要把所有相关数据从远端 PostgreSQL 取到本地然后再进行聚合运算，这极大地影响了整体运算效率。

声明式表分区

此前的版本中，实现分区表功能需要借助表继承机制。表继承机制的问题在于，用户需要自行编写触发器来实现把数据分流到子表中的过程。PostgreSQL 10 引入了 `PARTITION BY` 语法，利用该语法，用户只需在创建表时附加 `PARTITION BY` 子句就可以自动实现表分区。此后向父表插入数据时，数据会被分流到子表中，这一切都是自动的，无须用户预先创建触发器。请参考 6.1.3 节以了解详情。

查询性能优化

该版本中对查询性能实现了多方面的优化。

支持创建多字段统计信息

`CREATE STATISTICS` 命令支持针对多个字段建立统计信息。具体请参见示例 9-18。

支持 `IDENTITY` 数据类型

新增支持 `IDENTITY` 自增字段类型，创建表和修改表结构时都可以使用。增加该类型是为了在设计表的自增字段时更加符合业界通行做法。具体请参见示例 6-2。

1.6.3 PostgreSQL 9.6中引入的新特性

PostgreSQL 9.6 发布于 2016 年 9 月，是 9.x 系列中的最新版本。

支持并行查询

9.6 版之前，PostgreSQL 并不能充分地利用系统中的多核能力。9.6 版中，PostgreSQL 引擎能够将一些特定类型的查询语句放到多个处理器核心上并行执行。支持并行执行的操作有：顺序扫描、部分 join 以及部分聚合操作。然而，增删改这些修改数据的操作无法实现并行。支持并行化的工作还在进行中，最终目标是所有的语句都能利用到多核并行执行。详情参见 9.4 节。

支持关联词组全文检索

全文检索支持关联词组搜索，可以使用 `<->` 运算符来表示两个关联词之间的距离。即使关联词没有连在一起出现，只要二者出现的位置不超出前述距离，也认为该关联词组搜索命中。在 9.6 之前的版本中，只能搜索一些单个的词，但该功能使得用户可以搜索一

组有前后顺序的词组。详情请参见 5.8 节。

psql 工具支持 \gexec 参数

该参数的功能是读取并执行一个根据查询语句动态生成的 SQL。详情请参见 3.4.5 节。

postgres_fdw 增强

对于简单的更新、插入和删除来说，速度有很大提升。请参考博文“Directly Modify Foreign Tables”以了解详情。

FDW 关联操作下推

包括 postgres_fdw 在内的部分 FDW 已支持该特性。当需要对多张外表做关联查询时，之前的做法是把每张外表的数据取回本地然后做关联计算，支持了该特性的 FDW 的优化处理方式是：如果需要关联的两张外表都来自于同一台外部服务器，那么就把 join 操作下推到这个外部服务器上去执行，然后仅把关联结果拿回来。这样可以极大地减少经网络传输的记录数。当关联操作可以过滤掉大量数据时，这个特性带来的性能提升是巨大的。

1.6.4 PostgreSQL 9.5中引入的新特性

PostgreSQL 9.5 发布于 2016 年 1 月，其主要的新特性如下。

外部表架构的改进

支持了新的 IMPORT FOREIGN SCHEMA 命令，通过该命令可以从外部服务器中加载表结构信息从而实现批量创建外表。支持外表继承：本地表可以继承自外表，外表也可以继承自本地表，外表也可以继承自另一张外表。支持在外表上创建约束。详情请参见 10.3 节和 10.3.3 节。

支持无日志表直接升级为日志表

由于无日志表不写日志，往其中导入数据时会很快，但缺点是数据库崩溃时会完全丢失数据。在此前的版本中，当数据完全导入无日志表后，要想把无日志表改为日志表需要创建一张新表，然后再把无日志表的数据写入新表。9.5 版提供了一个新的语法：ALTER TABLE ... SET UNLOGGED，可以直接把无日志表修改为日志表，这样就省去了重新建表并搬迁数据的过程。

array_agg 函数支持数组入参

array_agg 函数可以将某字段的多条记录聚合为一个数组。在 9.5 版之前，入参字段类型不能是一个数组，但 9.5 版之后支持输入数组入参，这样就可以构造出多维数组。详情请参见示例 5-17。

支持 BRIN 索引类型

BRIN（block range index）是一种新型的索引，相比 B-树和 GIN 索引有着更小的内存占用。有些情况下 BRIN 索引会比前述两种索引速度更快。详情请参见 6.3 节。

支持 GROUPING SETS、ROLLUP 和 CUBE 聚合语法

该特性用于聚合查询语句中，可以实现一次性获取多维度汇聚结果。具体例子请参见 7.7 节。

仅索引扫描

如果要查询的字段在索引中已经全部包含，则仅扫描索引即可获得查询结果，而不用访问表数据，这种查询模式称为“仅索引扫描”。当前只有 GiST 索引支持此特性。

支持“无则插入有则更新”处理（即 `UPSERT` 能力）

9.5 版之前，如果插入或者更新操作违反了主键约束或者其他约束，那么该操作会失败。9.5 版提供了一种机制，允许用户捕获该异常并自定义替代操作或者跳过引发问题的记录。详情请参见 7.2.11 节。

支持跳过无法写锁定的记录

如果用户希望锁定一些记录以用于后续更新，那么可以使用 `SELECT ... FOR UPDATE` 来锁定这些记录。在 9.5 版之前，如果试图锁定的部分或者全部记录已被其他用户锁定，那么当前用户的锁定操作就会报错。9.5 版中提供了一个 `SKIP LOCKED` 子句语法，可以跳过那些已经被别人锁定的记录，这样就不会报错。

行级安全控制

支持对用户设置记录级的读写权限，即同一张表中，部分记录只允许某个用户读写，另外一部分记录只允许另外一个用户读写。在多租户场景下，显然该功能是特别有用的；对于需要进行访问权限隔离而又无法通过分表实现的场景，该功能也特别有价值。

1.6.5 PostgreSQL 9.4中引入的新特性

9.4 版发布于 2014 年 9 月，主要包含以下新特性。

物化视图特性的改进

在 9.3 版中，刷新物化视图期间会对其加锁并禁止访问，而加锁时间可能会比较长，这直接导致在生产环境中物化视图的实用价值严重受限。9.4 版中取消了刷新时的加锁动作，因此即使是正在被刷新的物化视图也可被访问。但请注意：利用此特性的前提是物化视图必须拥有一个唯一索引。

新增支持用于计算百分比的分析函数

新增了对 `percentile_disc`（不连续百分比）和 `percentile_cont`（连续百分比）这两个分析函数的支持，须配合 `WITHIN GROUP (ORDER BY...)` 子句使用。PostgreSQL 专家 Hubert Lubaczewski 在文章“Ordered Set Within Group Aggregates”中介绍了 `ORDERED SET WITHIN GROUP` 聚合运算。在 9.5 之前的版本中，PostgreSQL 并未提供过计算中位数的函数。这是有原因的。如果你对计算中位数的相关算法有所了解的话，应该知道这种算法中最后都会有一个额外的类似于“平局加时赛”的计算步骤，这个步骤的存在使得将中位数算法实现为聚合函数非常困难。9.5 版中引入的这两个分析函数使用了一种快速中位数估算算法，从而绕开了前述问题。我们将在 7.2.16 节对这两个函数进行更深入的介绍。

支持对视图更新操作的结果范围进行限制

创建视图时支持 `WITH CHECK OPTION` 子句，其作用是确保在视图上执行更新或者插入操作时，修改后或者新插入的记录仍然在本视图可见范围内。详情请参见示例 7-3。

新增对 JSONB 数据类型的支持

该数据类型是 JSON（JavaScript Object Notation）类型的二进制存储版本。通过 JSONB 类型可以对 JSON 格式的文档数据建立索引，并可加快对其内部元素的访问速度。详细信息请参考 5.6 节，同时可参考这两篇博客文章：“Introduce jsonb: A Structured Format for Storing JSON”以及“JSONB: Wildcard Query”。

GIN 索引改进

GIN 索引在设计时已经考虑了要适用于全文搜索、三连词处理、`hstore` 键值数据库以及 `jsonb` 类型支持等场景。在很多情况下你甚至可以把它当作 B- 树索引的替代品，一般来说 GIN 索引与 B- 树的查找效率相当，但占用空间更少。9.5 版中，GIN 索引的查询速度被进一步提升。详情请参见“GIN as a Substitute for Bitmap Indexes”这篇文章的介绍。

支持更多 JSON 函数

具体包括 `json_build_array`、`json_build_object`、`json_object`、`json_to_record` 和 `json_to_recordset` 等几个函数。

加速跨表空间的对象搬迁

支持使用以下语法轻松地将所有数据库对象从一个表空间移动到另一个表空间：`ALTER TABLESPACE old_space MOVE ALL TO new_space;`。

支持对返回的结果集中的记录加上数字编号

现在可以为查询结果中的记录添加行号。行号用系统字段 `ordinality`（该字段是在 ANSI SQL 标准中定义的）表示。当需要将存储为数组、`hstore` 键值对或者复合类型的非格式化数据转换为格式化记录时，该功能特别有用。以下是一个使用 `hstore` 的例子：

```
SELECT ordinality, key, value
 FROM each('breed=>pug,cuteness=>high'::hstore) WITH ordinality;
```

支持通过执行 SQL 命令来更改系统配置设置

利用 `ALTER system SET ...` 语法，无须修改 postgresql.conf 文件即可设置全局系统配置。postgresql.conf 文件具体的修改方法请参见 2.1.2 节。这也意味着用户可以通过编程的方式动态地调整系统配置项，但请注意：有的配置项修改后需要重启数据库才能生效。

触发器功能增强

9.4 版中支持了对外部表创建触发器。

数据行转列能力增强

`unnest` 函数用于实现数据的行转列操作，即将一个横向的数组转换为纵向的一个字段的多行记录。该函数可接受多个数组作为入参，每个数组转换后成为单独的一列，即输入几个数组转换后就有几列。如果每个数组的元素个数不一样，9.4 版之前转换后的结果是不可预知的，9.4 版本后这种情况下转换的结果是可预知的，会以最长的数组为标准，其他不足此长度的数组元素补 null。

新增 `ROWS FROM` 语法

该语法可以将多个函数返回的结果集逐行拼接起来，最后作为一个完整的结果集返回，

因此即使这些结果集之间的元素个数不一致也没关系，如下例所示：

```
SELECT *
FROM ROWS FROM (jsonb_each('{"a":"foo1","b":"bar"}'::jsonb),
                              jsonb_each('{"c":"foo2"}'::jsonb))
                              x (a1,a1_val,a2_val);
```

支持动态启用后台工作线程

当使用 SQL 或 PostgreSQL 函数都无法实现所需要的功能时，可以使用 C 语言编码实现动态后台工作线程来达成目标。9.4 版源码的 contrib/worker_spi 目录下实现了一个小型的示例，可供参考。

1.7 数据库驱动程序

任何情况下，你都不可能脱离具体的业务系统而仅仅使用 PostgreSQL 数据库本身，那显然是无意义的。为了实现 PostgreSQL 与业务系统之间的交互，就需要借助数据库驱动程序。PostgreSQL 拥有大量免费驱动，支持各种编程语言和开发工具。此外，很多商业公司也以很低廉的价格提供了各有特色的驱动。目前比较流行的几种开源驱动如下。

- PHP 驱动：PHP 语言广泛应用于 Web 开发领域，大多数 PHP 发行包都自带了较老的 pgsql 驱动或者是较新的 `pdo_pgsql` 驱动。一般来说这两种驱动默认都会安装，不过可能需要修改 php.ini 来决定启用哪一种。
- JDBC 驱动：Java 开发所使用的 JDBC 驱动一直是与最新版 PostgreSQL 同步更新的，可以从 PostgreSQL 官方站点下载。
- .NET 驱动：.NET 框架（含微软的官方版和 Mono 社区的开源版）可使用 Npgsql 驱动。目前该驱动支持微软 .NET 框架，包括微软 Entity Framework 开发框架以及 Mono 开源 .NET 框架。
- ODBC 驱动：如果需要从微软 Access、Excel 或者其他支持 ODBC 的产品连接到 PostgreSQL，可从 PostgreSQL 官网下载 ODBC 驱动，支持 32 位和 64 位两个版本。
- LibreOffice/OpenOffice 驱动：LibreOffice 3.5 及之后的版本中自带了 PostgreSQL 驱动，但 3.5 之前的版本以及 OpenOffice 是不带的，可以使用 JDBC 或者 SDBC 驱动。更多细节请参见“OO Base and PostgreSQL”这篇博文。
- Python 驱动：Python 可通过多种驱动访问 PostgreSQL，目前 psycopg2 是最流行的一种。Python 的 Django 开发框架对 PostgreSQL 也有着良好的支持。如果你需要一个关系－对象映射工具（即通常所说的 ORM 工具），可以考虑使用最广泛的 SQL Alchemy 工具，著名的外部数据源封装器开发平台 Multicorn 内部就使用了它。
- Ruby 驱动：对 Ruby 开发人员来说，请使用 rubygems pg 驱动。
- Perl 驱动：Perl 可以使用 DBI 和 DBD::Pg 驱动。也可以使用由 CPAN 网站提供的 DBD::PgPP 驱动。
- Node.js 驱动：Node.js 是一个 JavaScript 框架，可用于构建可扩展的网络应用。该平台目前支持两种 PostgreSQL 驱动：一种是 Node Postgres，该驱动可以选择是否绑定本地 libpq 库，而且基于纯 JavaScript（无须编译）；另一种是 Node-DBI。

1.8 如何获得帮助

在使用 PostgreSQL 的过程中，你迟早会需要寻求帮助，而且这一天往往会比预期来得早。我们希望你能够尽早了解到求助的途径。我们最为推荐的途径是邮件列表，不管你是 PostgreSQL 的新用户还是老用户，邮件列表都能为你解答技术问题。可以先打开 PostgreSQL 邮件列表页面。如果你是新手，那么订阅 PGSQL-General 这个邮件列表是最合适的。如果你认为自己发现了 PostgreSQL 的 bug，那么打开“PostgreSQL 故障报告”这个页面，上面会告诉你具体如何操作。

1.9 PostgreSQL的主要衍生版本

PostgreSQL 使用了 MIT/BSD 风格的许可证，任何人都可以合法地对其修改并二次传播，因此对于那些想创建自己数据库分支的人来说，PostgreSQL 是绝佳的选择。在过去的很多年间，有很多团队创建了自己的 PostgreSQL 分支版本，并且对社区也做出了相应的回馈，有的把自己的修改贡献回了 PostgreSQL 的主干代码，有的对社区给予了资金支持。访问 https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases 这个地址，就可以看到 PostgreSQL 数据库的所有衍生产品。

很多流行的分支版本是商业化的闭源软件。比如目前数据仓库领域使用很广泛的 Netezza 就是源自 PostgreSQL。亚马逊公司的 Redshift 数据仓库事实上是 PostgreSQL 的一个分支的分支。亚马逊还有其他两个与原生 PostgreSQL 血缘关系较近的产品：Amazon RDS for PostgreSQL 和 Amazon Aurora for PostgreSQL。这两个产品会与 PostgreSQL 开源版本的主干代码保持同源，并确保与原生 PostgreSQL 提供完全相同的 SQL 语法，同时额外提供了更强的管理功能并在速度方面做了一些优化。EnterpriseDB 公司推出的 PostgreSQL Advanced Plus 也是以 PostgreSQL 为基础，另外增加了对于 Oracle 语法和特性的兼容支持，以吸引原 Oracle 用户。EnterpriseDB 公司向 PostgreSQL 社区提供了资金和开发力量的支持，对此我们表示感谢。他们的 Postgres Plus Advanced Server 产品在版本更新节奏上也一直是密切跟进最新的 PostgreSQL 稳定版的。

Postgre-X2、Postgres-XL 和 GreenPlum 是三款还处于发展初期的 PostgreSQL 开源衍生产品，其中 GreenPlum 曾经有一段时间是闭源的。这三款产品的目标都是处理大规模数据分析和复制工作。

PostgreSQL 之所以衍生版本众多，部分原因是主干版本对于一些小众的需求可能不会及时支持，另外主干版本的发布节奏也不能满足所有人的要求，那么自己拉出来一个分支版本提前进行修改和测试就是更好的选择。很多这种分支版本中开发出来的新特性最终都汇合到了主干，比如 2nd Quadrant 公司支持多主和双向复制特性的 BDR 产品分支中的逻辑复制功能就被汇合入了主干，用于强化 PostgreSQL 原生的复制功能。PostgreSQL-XL 中开发的一些并行化特性将来也可能会合入 PostgreSQL 主干中。

Citus 是一个支持实时大数据处理和并行查询功能的 PostgreSQL 分支，从 PostgreSQL 9.5 开始它被改造成了 PostgreSQL 的一个扩展包，使用起来更加方便。

Google 最近发布了它的 Google Cloud SQL for PostgreSQL 产品，目前还处在 beta 测试阶段。

第2章

数据库管理

本章涵盖了管理 PostgreSQL 服务器需要了解的一些基本操作，包括角色与权限管理、database 的创建、安装扩展包、数据备份与恢复等。在开始之前，请先安装好 PostgreSQL 及其相应的管理工具，并且要保证自己有权任意地调整和使用该套环境。

2.1 配置文件

配置文件控制着一个 PostgreSQL 服务实例的基本行为，主要包含以下三个文件。

postgresql.conf
: 该文件包含一些通用设置，比如内存分配、新建 database 的默认存储位置、PostgreSQL 服务器的 IP 地址、日志的位置以及许多其他设置。

pg_hba.conf
: 该文件用于控制 PostgreSQL 服务器的访问权限，具体包括：允许哪些用户连接到哪个数据库，允许哪些 IP 地址连接到本服务器，以及指定连接时使用的身份验证模式。

pg_ident.conf
: 如果该文件存在，则系统会基于文件内容将当前登录的操作系统用户映射为一个 PostgreSQL 数据库内部用户的身份来登录。有些人会把操作系统的 root 用户映射为 PostgreSQL 的 `postgres` 超级用户账号。



在 PostgreSQL 的官方术语体系中，“角色”（role）就表示“用户”（user），但并不是所有的角色都需要具备登录权限，比如组角色（group role）通常就不需要。为了便于理解，本书中我们还是使用“用户”一词，它的含义就是具备登录权限的角色。

如果你在安装过程中使用了默认配置，则上述文件会位于 PostgreSQL 主数据文件夹中。你可以使用任何文本编辑器来编辑这些文件，或者在 pgAdmin 的 Admin Pack 页面上修改，详情请参见 4.2.3 节。如果你不确定这些文件的具体位置，以超级用户身份连接到任何一个数据库上并执行示例 2-1 中的查询语句即可找到。

示例 2-1　配置文件的位置

```
SELECT name, setting FROM pg_settings WHERE category = 'File Locations';

       name        |                 setting
-------------------+------------------------------------------
 config_file       | /etc/postgresql/9.6/main/postgresql.conf
 data_directory    | /var/lib/postgresql/9.6/main
 external_pid_file | /var/run/postgresql/9.6-main.pid
 hba_file          | /etc/postgresql/9.6/main/pg_hba.conf
 ident_file        | /etc/postgresql/9.6/main/pg_ident.conf
(5 rows)
```

2.1.1　让配置文件生效

有些配置项修改后需要重启 PostgreSQL 服务实例才能生效，重启时会关闭所有客户端连接。有的配置项只需重新加载一下配置文件即可生效，此后连接上来的新用户都会自动读取到新的配置。重新加载配置文件时，原来已连接的用户会话不会受到影响。如果你不确定修改了某个配置后是否需要重启，请查看下该配置项的 `context` 属性，如果是 `postmaster`，那么需要重启；如果是 `user`，那么重新加载配置文件即可。

1. 重新加载配置文件

有若干种方法可以重新加载配置文件，其中一种是打开控制台窗口并执行以下命令：

```
pg_ctl reload -D 你的数据目录
```

如果你在 RedHat Enterprise Linux、CentOS 或者是 Ubuntu 中以服务的形式安装了 PostgreSQL，那么执行以下命令即可：

```
service postgresql-9.5 reload
```

命令中的 `postgresql-9.5` 是你的服务名。（有的老版本的 PostgreSQL 的服务名就是 `postgresql`，不带版本号。）

还有一种加载配置文件的方法是以超级用户身份登录到任何数据库并执行以下 SQL：

```
SELECT pg_reload_conf();
```

最后还有一种方法是用 pgAdmin 工具实现重加载，详情参见 4.2.3 节。

2. 重启PostgreSQL运行实例

一些底层的配置修改后必须重启 PostgreSQL 服务实例才能生效。可以通过先停止再启动 PostgreSQL 后台服务的方法来完成此过程。将服务器直接断电重启当然也可以（开玩笑，尽量别这么干）。

PostgreSQL 运行实例无法通过执行 PostgreSQL 本身的命令行来实现重启，只能通过操作

系统的命令来实现。在 Linux/Unix 系统上，如果 PostgreSQL 实例是以服务形式运行的，请执行：

```
service postgresql-9.6 restart
```

如果不是以服务形式运行的，请执行：

```
pg_ctl restart -D 你的数据目录
```

在 Windows 上，请打开服务管理器，找到 PostgreSQL 服务，然后对它点击重启即可。

2.1.2 postgresql.conf

postgresql.conf 文件包含了 PostgreSQL 服务正常运行所必需的基础设置。你可以在数据库级、用户级、会话级甚至是函数级重载这些设置。“Tuning Your PostgreSQL Server”这篇文章详细介绍了如何通过修改设置来优化你的 PostgreSQL 系统。

PostgreSQL 9.4 引入了一个重要的变更：该版本引入了一个新的名为 postgresql.auto.conf 的配置文件，其中的配置项会覆盖 postgresql.conf 的同名配置项。建议你不要直接修改 postgresql.conf，而是优先修改 postgresql.auto.conf。

1. 查看postgresql.conf中的配置

通过查询 `pg_settings` 视图即可查看所有配置项内容，无须打开配置文件。示例 2-2 展示了具体的查询语法。

示例 2-2　关键的设置

```
SELECT
    name,
    context ❶,
    unit ❷,
    setting, boot_val, reset_val ❸
FROM pg_settings
WHERE name IN ( 'listen_addresses','deadlock_timeout','shared_buffers',
    'effective_cache_size','work_mem','maintenance_work_mem')
ORDER BY context, name;

name                 | context    | unit | setting | boot_val  | reset_val
----------------------+------------+------+---------+-----------+-----------
listen_addresses     | postmaster |      | *       | localhost | *
shared_buffers       | postmaster | 8kB  | 131584  | 1024      | 131584
deadlock_timeout     | superuser  | ms   | 1000    | 1000      |
effective_cache_size | user       | 8kB  | 16384   | 16384     | 16384
maintenance_work_mem | user       | kB   | 16384   | 16384     | 16384
work_mem             | user       | kB   | 5120    | 1024      | 5120
```

❶ `context` 字段代表配置项的作用范围。各配置项的作用范围大小不一，具体取决于 `context` 字段的内容。

`context` 值为 `user` 表示是用户级配置项，它可以被每个用户单独修改，也就是说该配置项针对每个用户都可以有不同的值，用户修改后会在自己的所有会话中生效。如果是超级用户修改了一个 `user` 级的配置项，那么所有此后连接上来的用户都会将这个修改过的值作为默认值。

context 值为 superuser 表示是超级用户级配置项，只能由超级用户来修改，修改并且重新加载后会在所有用户会话中生效。非超级用户不能在自己的会话中修改覆盖这个值。

context 值为 postmaster 表示是整个服务实例级配置项（postmaster 就代表了 PostgreSQL 服务实例），更改后需要重启 PostgreSQL 服务才能生效。

context 为 user 和 superuser 的配置项可以在 database 级、用户级、会话级和函数级分别设置。比如对于一个能写出非常烧脑的 SQL 的专家用户来说，work_mem 参数应该设置得大一些。又比如，如果有一个函数里面会进行密集的排序操作，那么可以仅针对此函数调大 work_mem 的值。database 级、用户级、会话级和函数级的参数设置不需要执行重加载操作。数据库级的参数设置会在用户下次连接到该数据库时生效。会话和函数级的参数设置立即生效。

❷ 请注意内存相关参数所使用的单位。在示例 2-2 的输出结果中，内存相关参数有些以 8KB 为单位，有些以 KB 为单位。不管最终显示时用的是什么单位，设置时都可以用任何你觉得方便的单位。对于大多数内存相关配置项来说，128MB 是一个比较合适的值。我们认为以 8KB 为单位来显示内存配置是一种很糟糕的做法，看起来很不直观，也容易改错。可以用 SHOW 命令来查看配置项，它的输出结果会自动根据数值大小选择合适的单位来呈现，比较直观。例如：执行 SHOW shared_buffers; 显示的结果是 1028MB（而非前面查出来的 131 584 个 8KB）。类似地，运行 SHOW deadlock_timeout; 返回的是 1s（而非前面查出来的 1000 ms）。如果你想以合适的单位查看所有参数，请执行 SHOW ALL。

❸ setting 是指当前设置；boot_val 是指默认设置；reset_val 是指重新启动服务器或重新加载设置之后的新设置。修改了设置后，一定要记得查看一下 setting 和 reset_val 并确保二者是一致的，否则说明设置并未生效，需要重新启动服务器或者重新加载设置。

9.5 版中引入了一个新的 pg_file_settings 视图，通过该视图也可以进行配置信息查询。查询该视图会列出每个配置项所属的配置文件。其中 applied 字段表示该配置项是否已经生效，如果值为 f，表示需要重启服务器或者重加载配置文件。如果 postgresql.conf 和 postgresql.auto.conf 中存在同名配置，那么后者会覆盖前者，前者在 pg_file_settings 中对应的条目会显示 applied 字段为 f。具体如示例 2-3 所示。

示例 2-3　查询 pg_file_settings 视图

```
SELECT name, sourcefile, sourceline, setting, applied
FROM pg_file_settings
WHERE name IN ('listen_addresses','deadlock_timeout','shared_buffers',
    'effective_cache_size','work_mem','maintenance_work_mem')
ORDER BY name;

name                 | sourcefile                     | sourceline | setting | applied
---------------------+--------------------------------+------------+---------+--------
effective_cache_size | E:/data96/postgresql.auto.conf|         11 | 8GB     | t
listen_addresses     | E:/data96/postgresql.conf     |         59 | *       | t
maintenance_work_mem | E:/data96/postgresql.auto.conf|          3 | 16MB    | t
shared_buffers       | E:/data96/postgresql.conf     |        115 | 128MB   | f
shared_buffers       | E:/data96/postgresql.auto.conf|          5 | 131584  | t
```

请特别注意 postgresql.conf 或者 postgresql.auto.conf 中的以下网络设置，如果设得不对会导

致客户端无法连接，修改这些值是一定要重新启动数据库服务的。

`listen_addresses`

表示 PostgreSQL 服务使用的 IP 地址，一般会设置为 `localhost`，表示本机的 IPV6 或者 IPV4 地址。但也有很多人会设置为 `*`，表示使用任意本机 IP 地址均可连接到 PostgreSQL 服务。

`port`

PostgreSQL 服务的侦听端口，默认值为 5432。这个端口广为人知，因此建议修改为别的端口，这样可以降低受攻击的概率。当然，如果你在同一台机器上启动了多个 PostgreSQL 服务实例，那么每个服务实例的侦听端口都不能重复，此时也需要修改该配置。

`max_connections`

系统允许的最大并发连接数。

`log_destination`

这个配置项的名字可能有点误导，它实际指的是日志文件的输出格式而非输出的物理位置。默认值是 `stderr`。如果你希望保存日志内容以做进一步分析，建议把它修改为 `csvlog`，这样就更容易将日志导入第三方日志分析工具中使用。注意，如果希望记日志的话，请一定要将 `logging_collection` 配置项设为 `on`。

下面介绍的几个配置项会影响整体系统性能，其默认值一般都不是最优的。建议用户对它们有所了解后尽快根据实际情况做优化调整。

`shared_buffers`

此设置定义了用于缓存最近访问过的数据页的内存区大小，所有用户会话均可共享此缓存区。此设置对查询速度有着重大影响，一般来说设置得越大越好，至少应该达到系统总内存的 25%，但不宜超过 8GB，因为超过后会出现“边际收益递减”效应，即消耗的内存很多，但得到的速度提升却很少，得不偿失。修改此设置需要重启 PostgreSQL 服务。

`effective_cache_size`

该配置项是一个估算值，表示操作系统分配多少内存给 PostgreSQL 专用。系统并不会根据这个值来真实地分配这么多内存，但是规划器会根据这个值来判断系统能否提供查询执行过程中所需的内存。如果将此设置设得过小，远远小于系统的真实可用内存量，那么可能会给规划器造成误导，让规划器认为系统可用内存有限，从而选择不使用索引而是执行全表扫描（因为使用索引虽然速度快，但需要占用更多的中间内存）。在一台专用于运行 PostgreSQL 数据库服务的服务器上，建议将 `effective_cache_size` 的值设为系统总内存的一半或者更多。此设置的更改可动态生效，执行重新加载即可。

`work_mem`

此设置指定了用于执行排序、散列关联、表扫描等操作的最大内存大小。要得到此设置的最优值需要考虑以下因素：数据库的使用方式，需要预留多少内存给除数据库系统外的程序，以及服务器是否专用于运行 PostgreSQL 服务等问题。如果使用场景仅仅是有很多用户并发执行简单查询，那么这个值可以设得小一点，这样每个用户都得以较为公平地使用内存，否则第一个用户就可能把内存占光。这个值该设多大同样取决于你的机

器上总共有多少内存可用。关于 `work_mem` 设置有一篇很好的文章“Understanding work_mem”。此设置的更改可动态生效，执行重新加载即可。

`maintenance_work_mem`
: 此设置指定了可用于 vaccum（即清空已标记为“被删除”状态的记录）这类系统内部维护操作的内存总量。其值不应大于 1GB。此设置的更改可动态生效，执行重新加载即可。

`max_parallel_workers_per_gather`
: 这是 9.6 版中新引入的一个配置项，用于控制语句执行的并行度。该配置项决定了执行计划的每个 gather 节点中最多允许启动多少个 worker 进程并行工作。默认值为 0，表示不启用并行功能。如果你的 PostgreSQL 服务器是多核的，那么建议评估后尝试开启此参数。并行处理是 9.6 版开始支持的新特性，因此你需要做一些测试来验证到底将并行度设为多大时效果最好。另外，注意该配置项的值应该小于 `max_worker_processes` 的值（默认为 8），因为用于并行处理的后台 worker 进程是系统总体后台 worker 进程的一部分。

  版本 10 中引入了一个新的参数叫作 `max_parallel_workers`，用于控制所有后台 worker 进程中有多少可用于并行。

2. 修改postgresql.conf中配置项的值

PostgreSQL 9.4 中引入了对新的 `ALTER SYSTEM` SQL 命令的支持，使用该命令可以更改设置。例如，如果要设置一个全局生效的 `work_mem`，执行以下命令即可：

```
ALTER SYSTEM SET work_mem = '500MB';
```

该命令不会直接修改 postgresql.conf 文件本身，而是会去修改 postgresql.auto.conf 文件。

每个设置有着各自不同的特性，有的更改后必须重启数据库服务才能生效，有的只要重新加载一次就可以了，下面这个命令可以实现设置重新加载：

```
SELECT pg_reload_conf();
```

如果你需要时常修改很多配置项，那么可以尝试将它们分门别类存放到多个配置文件中，然后通过在 postgresql.conf 中使用 `include` 或者 `include_if_exists` 前缀来引入这些配置文件。具体语法如下：

```
include '配置文件名'
```

这里的配置文件名可以是绝对路径也可以是相对路径，相对路径的起始位置就是 postgresql.conf 文件本身所在的位置。

3.“我修改了postgresql.conf文件，结果数据库服务无法启动了，该怎么办？”

定位这种问题最简单的方法是查看日志文件，该文件位于 PostgreSQL 数据文件夹的根目录或者 pg_log 子文件夹下。只要找到最近修改的那个日志文件并查看其最后一行的内容，就能找到本次问题的相关错误日志。日志的内容一般都是比较直白易懂的，你看了就会明白。

最常见的错误是把 `shared_buffers` 设得太大了。还有一个常见问题是由于上次系统异常关闭导致遗留了一个没来得及删除的 postmaster.pid 文件，该文件就位于数据文件夹下，你可以手动删除该文件并重新启动 PostgreSQL。

2.1.3 pg_hba.conf

pg_hba.conf 文件指定了哪些 IP 地址和哪些用户可以连接到 PostgreSQL 数据库，同时还规定了用户必须使用何种身份验证方式登录。针对该文件的修改可动态生效，执行一次配置重加载即可。一个典型的 pg_hba.conf 文件看起来如示例 2-4 所示。

示例 2-4　pg_hba.conf 文件示例

```
# TYPE   DATABASE USER ADDRESS         METHOD
host     all      all  127.0.0.1/32    ident ❶
host     all      all  ::1/128         trust ❷
host     all      all  192.168.54.0/24 md5 ❸
hostssl  all      all  0.0.0.0/0       md5 ❹

# TYPE DATABASE    USER     ADDRESS      METHOD
# Allow replication connections from localhost,
# by a user with replication privilege. ❺
#host  replication postgres 127.0.0.1/32 trust
#host  replication postgres ::1/128      trust
```

❶ 身份验证模式。一般有以下几种常用选项：`ident`、`trust`、`md5`、`peer` 以及 `password`。

❷ 用于定义 IPv6 网段。只有服务器支持 IPv6 时才可以配置该项，如果在非 IPv6 网络环境下配置了这样的条目，会导致 pg_hba.conf 文件无法加载，从而进一步导致任何客户端都无法连接。

❸ 用于定义 IPv4 网段。第一部分是网络地址，后面跟着的是子网掩码，比如 192.168.54.0/24。这样定义的效果是该子网中的任何客户端都可以连到本 PostgreSQL 实例。

❹ 这是针对 SSL 连接的规则。本例中，任何能使用 SSL 方式连接的客户端都可以连接到本 PostgreSQL 实例。

SSL 相关配置都在 postgresql.conf 和 postgresql.auto.conf 中，包含以下几项：`ssl`、`ssl_cert_file`、`ssl_key_file`。一旦确认客户端支持 SSL，PostgreSQL 服务端就会接受其连接请求，并且该连接上所有传输的内容都会使用 ssl key 加密。

❺ 这是允许与本节点构成复制关系的其他 PostgreSQL 服务器节点的 IP 网段。

对于每一个连接请求，`postgres` 服务会按照 pg_hba.conf 文件中记录的规则条目自上而下进行检查。当匹配到第一条允许此请求接入的规则时，就不再往下检查，系统将允许该连接请求。类似地，如果匹配到一条拒绝此连接请求的规则，也不再继续检查，并拒绝该连接请求。如果一直搜索到文件的末尾都没能找到匹配项，那么该连接请求会被拒绝。大家常犯的一个错误是把规则的顺序放错。例如，如果你将 `+0.0.0.0/0 reject+` 规则放到 `+127.0.0.1/32 trust` 的前面，那么此时本地用户全都无法连接，即使下面有规则允许也不行。

PostgreSQL 10 中引入了一个新的名为 `pg_hba_file_rules` 的视图，通过查询该视图可以直

接看到 pg_hba.conf 中的内容。

1. “我修改了pg_hba.conf文件，结果服务器崩溃了，该怎么办？”

不用担心，这种事情经常发生，解决起来不难。这一般是因拼写错误或增加了一种不支持的身份验证模式导致的。如果 `postgres` 服务无法正确地解析 pg_hba.conf 文件，那么为确保系统安全，它会禁止所有的连接请求甚至是禁止系统启动。最简单的诊断方法是看一下日志，文件就在数据文件夹的根目录下或者其 pg_log 子文件夹下。可以打开修改日期最近的日志文件并看一下最后一行的内容，错误提示信息一般就在那里，而且一般都是很好理解的。如果你经常会笔误，改错东西，那么请一定记得在修改配置文件之前做个备份。

2. 身份验证方法

PostgreSQL 提供了多种模式用于用户身份验证，很可能是所有数据库里面支持的模式最多的。大多数人只会用到其中几种最常见的：`trust`、`peer`、`ident`、`md5` 和 `password`。还有一种 `reject` 模式，其作用是拒绝所有请求。不过别忘了 pg_hba.conf 中还可以定义其他不同层面的身份验证模式（比如 `gss`、`sspi`、`ldap`、`radius`、`cert`、`pam` 等），它就好像是整个 PostgreSQL 服务器的看门人，确保整个系统的外部访问安全。当然，在通过了这一层外部安全控制并成功建立连接以后，连上来的用户仍需遵守角色权限和数据库访问限制等内部约束规则。

最常用的身份验证方法有以下这些。

`trust`

　　这是最不安全的身份验证模式，用户无须提供密码就可以连接到数据库。只要源端 IP 地址、连接用户名、要访问的 database 名都与该条规则匹配，用户就可以连上来。`trust` 模式很不安全，因此应对其使用予以限制，即只能允许从数据库服务器本机发起的连接或者是同属内网的用户发起的连接使用此模式。但即使加了前述限制也不能保证安全，因为会有人通过伪装 IP 地址的方式来冒用此权限，所以有些安全意识比较强的人认为该模式应该被彻底禁用。然而在单用户的桌面环境下，这却是最常用的身份验证模式，因为一般这种场景下系统的安全性根本不是问题。

`md5`

　　该模式很常用，要求连接发起者携带用 md5 算法加密的密码。

`password`

　　该模式要求连接发起者携带明文密码进行身份验证。

`ident`

　　该身份验证模式下，系统会将请求发起者的操作系统用户映射为 PostgreSQL 数据库内部用户，并以该内部用户的权限登录，且此时无须提供登录密码。Windows 上不支持 `ident` 验证方式。

`peer`

　　该模式下系统会直接从操作系统内核获取当前连接发起者的操作系统用户名，如果与其请求连接的 PostgreSQL 用户名一致，即可连接成功。该模式仅可用于 Linux、BSD、Mac OS X 和 Solaris，并且仅可用于本地服务器发起的连接。

`cert`

该模式要求客户端必须使用 SSL 方式进行连接。连接发起者必须提供一个合法的 SSL 证书。该模式使用一个身份认证文件（比如 pg_ident）来将 SSL 证书映射为 PostgreSQL 数据库的内部用户，该模式在所有支持 SSL 连接的平台上都可用。

另外还有一些不常见的验证方式，比如 `gss`、`radius`、`ldap` 和 `pam`。有的可能不会默认安装。

多种身份验证模式是可以同时使用的，即使是针对同一个 database 也可以这么做，也就是说我们可以针对同一个 database 设置多条身份验证规则，并且每条规则的身份验证模式都不一样。但请你务必牢记 PostgreSQL 对于 pg_hba.conf 中的规则的查找顺序是从上到下，第一条匹配到的规则就是系统使用的规则。

2.2 连接管理

我们可能时不时地会遇到一些想要终止数据库连接的情况，比如有人执行了写得很糟糕的 SQL 语句把系统资源耗光，当然这肯定不是他的本意，又比如你在执行某些语句时发现其耗时太长，超出了自己的忍耐极限。发生这些情况时，我们一般都会希望结束这些操作或者干脆彻底终止这个连接。

强行中断语句执行或者终止连接是一种很不“优雅”的行为，应当尽量避免。在客户端应用程序中，首先应该采取措施防止或者想办法解决 SQL 语句出现失控（耗时长或者占资源多）的情况，然后再考虑通过在后台直接杀掉的方式处理。出于礼貌，你应该在终止连接之前通知相关用户其连接即将被强行终止，或者如果实在有必要，你也可以不管它什么礼貌不礼貌，等四下无人时直接终止这些连接就好了。

有的场景下我们会需要杀掉所有正在运行的更新操作，比如备份数据库之前以及恢复数据库之前。

要想终止正在执行的语句并杀掉连接，请使用以下步骤。

(1) 查出活动连接列表及其进程 ID。

```
SELECT  * FROM pg_stat_activity;
```

`pg_stat_activity` 视图包含每个连接上最近一次执行的语句、使用的用户名（`usename` 字段）、所在的 database 名（`datname` 字段）以及语句开始执行的时间。通过查询该视图可以找到需要终止的会话所对应的进程 ID。

(2) 取消连接（假设对应的进程号是 1234）上的活动查询。

```
SELECT  pg_cancel_backend(1234);
```

该操作不会终止连接本身。

(3) 终止该连接。

```
SELECT  pg_terminate_backend(1234);
```

有时候你会需要终止这个连接，特别是执行数据库恢复之前。如果仅仅是终止了正在执

行的语句而没有彻底杀掉连接，客户端是可以立即重新执行刚刚被终止掉的语句的，这又会导致系统陷入之前的状态。为了避免此种情况的发生，可以采用直接终止连接的方式。如果你未停止某个连接上正在执行的语句就直接终止该连接，那么这些语句也会被停止掉。

PostgreSQL 支持在 SELECT 查询语句中调用函数。因此，尽管 pg_terminate_backend 和 pg_cancel_backend 一次仅能处理一个连接，但你可以通过在 SELECT 语句中调用函数的方式实现一次处理多个连接。例如，如果你希望一次性终止某个用户的所有连接，那么可以执行以下语句。

```
SELECT pg_terminate_backend(pid) FROM pg_stat_activity
WHERE usename = 'some_role';
```

PostgreSQL 有一些语句参数可以用来控制语句的运行状态，一旦语句运行期间的某些状态值超过了这些运行参数所限定的范围，该语句会被系统自动杀掉。这些参数可以在服务实例级、database 级、用户级、会话级和函数级设置。参数值设为 0 代表禁用。

deadlock_timeout

该参数表示在进行死锁检测之前需要等待多久。[1] 默认是 1000 毫秒。如果你的业务系统中有大量更新操作，那么建议增大该值以减少死锁检测的次数。

与其依赖这个参数来解决死锁，我们更建议在 UPDATE 语句中加 NOWAIT 子句来避免死锁，例如：SELECT FOR UPDATE NOWAIT...。

该语句会在发生死锁时自动终止执行。

在 PostgreSQL 9.5 中，用户还有另一个选择：SELECT FOR UPDATE SKIP LOCKED，该语法可以跳过已经被别的用户锁定的记录。

statement_timeout

该参数可以控制一个语句能够执行的最长时间，超出限定的时间后该语句会被自动终止。该参数的默认值是 0，即无限制。如果你需要对一个运行可能耗时很久的函数加上最长运行时间限制，那么最好不要把这个参数设置为全局级别，仅在要控制的函数的定义中针对该函数自身设置一下即可。这样可以保证不会有“误杀”的情况发生，因为我们无法预料别的语句是否需要执行更长时间。如果函数因执行时间超长而被终止，那么调用该函数的事务也会跟着被回滚掉。

lock_timeout

该参数控制锁等待的最长时间，超出限定时间后等待锁的语句就会被自动终止。对于执行数据更新的语句来说，该参数有较大价值，因为每次更新数据之前都必须先获取待修改的记录上的排他锁，所以更新语句之间是最容易发生锁等待的。该参数的默认值为 0，表示不限制锁等待的时间。一般会在函数级或者会话级设置该参数。lock_timeout 的值应该设得比 statement_timeout 小，否则总会是语句先超时，这样 lock_timeout 就毫无意义了。

注 1：死锁检测是很昂贵的操作，因此系统不会每次发生锁等待时都做死锁检测。——译者注

`idle_in_transaction_session_timeout`

该参数表示一个事务可以处于 idle 状态的最长时间，超过限定的时间后该事务会被自动回滚。该参数的默认值为 0，表示事务可以永久处于 idle 状态。该参数是 9.6 版引入的，可以起到两个作用：防止一个空闲事务占着记录锁一直不释放从而阻塞别的事务继续运行，还可以防止一个数据库连接被一个空闲事务永远占用。

查看被阻塞语句的情况

从 9.1 版开始，`pg_stat_activity` 视图发生了较大变化，一些字段的名称发生了变化，一些字段被删除了，另外也新增了一些字段。从 9.2 版开始，该视图中原来的 `procpid` 字段被改名为 `pid`。

在 PostgreSQL 9.6 中，`pg_stat_activity` 视图中增加了更多关于等待锁的语句的信息。在之前的版本中，有一个名为 `waiting` 的布尔型字段，其值为 `true` 时表示该会话上有个语句正在等待别的语句释放资源，但看不出是在等待获取什么资源。9.6 版中，`waiting` 字段被取消，替代它的是两个名为 `wait_event_type` 和 `wait_event` 的字段，其中记录了当前会话上的语句在等待什么资源。因此在 9.6 版之前，可以使用 `waiting = true` 过滤条件查出所有被别人阻塞的语句；9.6 版之后，应更改为使用 `wait_event IS NOT NULL` 过滤条件来查找被阻塞的语句。

除了 `pg_stat_activity` 视图的结构发生了变化外，PostgreSQL 9.6 中还对一些之前版本中无法通过 `waiting = true` 过滤出来的锁等待进行了跟踪，这样用户就可以看到之前版本中无法查询出来的更轻量级的锁等待。如需了解 `wait_event` 字段所有可能的值，请参考官方手册中关于等待事件名和类型的说明。

2.3 角色

PostgreSQL 中使用**角色**（role）机制来处理用户身份认证。拥有登录数据库权限的角色称为**可登录角色**（login role）。一个角色可以继承其他角色的权限从而成为其**成员角色**（member role）；拥有成员角色的角色称为**组角色**[2]（group role）。（一个组角色可以是另一个组角色的成员角色，并且这种角色间的继承关系可以有无限多层，但除非你非常有把握能搞定这种多层嵌套关系，否则别这么干，因为你最后一定会把自己搞糊涂。）拥有登录权限的组角色称为**可登录的组角色**（group login role）。然而，基于安全性的考虑，数据库管理员一般不会为组角色授予登录权限，因为设计组角色的本意是将其作为一个“权限集合”使用，而不是将其作为一个真正需要登录权限的用户角色来使用。一个角色可被授予**超级用户**（SUPERUSER）权限，拥有此权限的角色可以彻底地控制 PostgreSQL 服务，因此授予这种权限时一定要慎重。

注 2：设计“组角色”这一功能的本意是为了将一组权限集中在一起成为一个“组”，然后便于以“组”为单位对这些权限进行管理，比如可以通过角色权限继承的方式一次性将这一组权限赋予其成员角色。——译者注



PostgreSQL 从最近的几个版本开始不再使用“用户”和“组”这两个术语。但你还会看到有人使用这两个术语，请记住“用户”和“组”分别代表“可登录角色”和“组角色”就好了。为保持前向兼容，`CREATE USER` 和 `CREATE GROUP` 这两个命令在当前版本中也是支持的，但最好不要使用它们，请使用 `CREATE ROLE`。

2.3.1　创建可登录角色

在 PostgreSQL 安装过程中的数据初始化阶段，系统会默认创建一个名为 `postgres` 的可登录角色（同时会创建一个名为 `postgres` 的同名 database）。你可以通过前面介绍过的 ident 或者 peer 身份验证机制来将操作系统的 root 用户映射到数据库的 `postgres` 角色，这样可以实现 root 用户免密登录，或者通过设置为 trust 模式的效果也是一样。数据库安装完成后，第一件要做的事就是用 psql 或者 pgAdmin 工具以 `postgres` 角色身份登录，然后创建其他已规划好的角色。pgAdmin 工具中有专门的图形界面用于创建角色，如果你希望用 SQL 语句手动创建，请参考示例 2-5 中的 SQL 语句。

示例 2-5　创建具备登录权限的角色

```
CREATE ROLE leo LOGIN PASSWORD 'king' VALID UNTIL 'infinity' CREATEDB;
```

`VALID` 子句是可选的，其功能是为此角色的权限设定有效期，如果不写则该角色永久有效。`CREATEDB` 子句表明为此角色赋予了创建新数据库的权限。

如果要创建一个具备超级用户权限的角色，可以参考示例 2-6。当然，要想创建一个超级用户，创建者自身也必须是一个超级用户。

示例 2-6　创建具备超级用户权限的角色

```
CREATE ROLE regina LOGIN PASSWORD 'queen' VALID UNTIL '2020-1-1 00:00' SUPERUSER;
```

前面两个例子中我们创建的都是可登录角色，如果需要创建不可登录的角色，省略掉 `LOGIN PASSWORD` 子句即可。

2.3.2　创建组角色

一般不应授予组角色登录权限，因为它是作为其他角色的容器而存在的。当然，这只是我们基于实践经验给出的建议，你也可以为组角色授予登录权限，这完全没问题。

可以用以下 SQL 创建组角色。

```
CREATE ROLE royalty INHERIT;
```

请注意关键字 `INHERIT` 的用法。它表示组角色 `royalty` 的任何一个成员角色都将自动继承其除“超级用户权限”外的所有权限。出于安全考虑，PostgreSQL 不允许超级用户权限通过继承的方式传递。如果不写 `INHERIT`，默认也会有 `INHERIT` 的效果，但为了清晰起见，建议还是写上。

如果希望禁止组角色将其权限授予成员角色，可以加上 `NOINHERIT` 关键字。

以下语句可以为组角色添加成员角色。

```
GRANT royalty TO leo;
GRANT royalty TO regina;
```

有些权限是无法被继承的，例如前面提过的 SUPERUSER 超级用户权限，然而成员角色可以通过 SET ROLE 命令来实现“冒名顶替”其组角色的身份，从而获得其父角色所拥有的 SUPERUSER 权限，当然这种冒名顶替的状态是有期限的，仅限于当前会话存续期间有效。举例如下。

我们先通过以下命令授予 royalty 组角色超级用户权限：

```
ALTER ROLE royalty SUPERUSER;
```

此时尽管 leo 是 royalty 的成员角色，也继承了其绝大多数的权限，但 leo 登录后依然不具备 SUPERUSER 权限。但执行以下语句即可获取 SUPERUSER 权限：

```
SET ROLE royalty;
```

请记住这种方法获取的 SUPERUSER 权限仅在会话存续期间有效。

虽然这个操作逻辑看起来有点怪异，但如果你不希望自己在登录后以 SUPERUSER 的身份犯下一些无可挽回的错误，那么这个方法值得考虑。

所有用户都可以使用 SET ROLE 这个命令，但还有一个比它更强大的命令：SET SESSION AUTHORIZATION，该命令只允许具备 SUPERUSER 权限的用户执行。为了便于理解，我们首先介绍 PostgreSQL 中的两个全局变量：current_user 和 session_user。执行以下语句即可查看这两个变量的值：

```
SELECT session_user, current_user;
```

首次登录成功后，这两个变量的值相同。执行 SET ROLE 会修改 current_user 的值，执行 SET SESSION AUTHORIZATION 会同时改变 current_user 和 session_user 的值。

以下是 SET ROLE 命令的主要特点。

- SET ROLE 无须 SUPERUSER 权限即可执行。
- SET ROLE 会修改 current_user 变量的值，但不修改 session_user 的值。
- 一个具备 SUPERUSER 权限的 session_user 同名角色可以通过 SET ROLE 设置为任何用户。
- 非超级用户可以通过 SET ROLE 设置为 session_user 同名角色或者其所属的组用户。
- SET ROLE 命令可以让执行角色获取到所“扮演”角色的全部权限，SET SESSION AUTHORIZATION 和 SET ROLE 权限除外。

SET SESSION AUTHORIZATION 是比 SET ROLE 更为强大的命令，其关键特性如下。

- 只有超级用户才可以执行 SET SESSION AUTHORIZATION。
- SET SESSION AUTHORIZATION 权限在整个会话存续期间都是有效的，也就是说即使超级用户通过 SET SESSION AUTHORIZATION 来“扮演”了一个非超级用户，只要会话未中断，都可以在上面再次执行 SET SESSION AUTHORIZATION 命令。
- SET SESSION AUTHORIZATION 会将 current_user 和 session_user 修改为要“扮演”的角色。

- 具备超级用户权限的 `session_user` 同名角色可以通过 `SET ROLE` 来“扮演”任何其他角色。

接下来我们通过一系列实验来演示 `SET ROLE` 和 `SET SESSION AUTHORIZATION` 之间的区别。请先以 leo 用户的身份登录，然后运行示例 2-7 中的代码。

示例 2-7　SET ROLE 和 SET AUTHORIZATION

```
SELECT session_user, current_user;

 session_user | current_user
--------------+--------------
 leo          | leo
(1 row)

SET SESSION AUTHORIZATION regina;

ERROR:  permission denied to set session authorization

SET ROLE regina;

ERROR:  permission denied to set role "regina"

ALTER ROLE leo SUPERUSER;

ERROR:  must be superuser to alter superusers

SET ROLE royalty;
SELECT session_user, current_user;

 session_user | current_user
--------------+--------------
 leo          | royalty
(1 row)

SET ROLE regina;

ERROR:  permission denied to set role "regina"

ALTER ROLE leo SUPERUSER;
SET ROLE regina;
SELECT session_user, current_user;

 session_user | current_user
--------------+--------------
 leo          | regina
(1 row)

SET SESSION AUTHORIZATION regina;

ERROR:  permission denied to set session authorization

-- 退出此会话然后重新以leo身份登录
SELECT session_user, current_user;
SET SESSION AUTHORIZATION regina;
SELECT session_user, current_user;
```

```
 session_user | current_user
--------------+--------------
 leo | leo
(1 row)
SET SESSION AUTHORIZATION
session_user | current_user
--------------+--------------
regina | regina
(1 row)
```

在示例 2-7 中，leo 用户不是超级用户，因此不能使用 `SET SESSION AUTHORIZATION` 命令。同样，他也不能通过 `SET ROLE` 来扮演 regina 角色，因为他不是 regina 组的成员。然而 leo 可以通过 `SET ROLE` 来扮演 royalty 角色，因为他是 royalty 组的成员（可以把 leo 理解成国王的伙伴）。扮演为 royalty 角色后，虽然 royalty 有超级用户权限，但他却不能再次扮演女王 regina，因为他并没有继承到 royalty 的超级用户权限，所以此时 `SET ROLE` 所能扮演的角色范围还是非超级用户 leo 所能扮演的范围。由于 royalty 组角色具备超级用户权限，因此他可以主动授予其成员角色 leo 超级用户权限。一旦 leo 被提权成为超级用户，他就可以扮演 regina 了。此时他可以使用 `SET SESSION AUTHORIZATION` 来完全获取 regina 用户的权限，`session_user` 和 `current_user` 也会都被设置为 regina。

2.4　创建database

一个不含任何附加子句的最简单 SQL 语句即可创建一个 database：

```
CREATE DATABASE mydb;
```

该命令会以 `template1` 库为模板生成一份副本并将此副本作为新 database。任何一个拥有 `CREATEDB` 权限的角色都能够创建新的 database。

2.4.1　模板数据库

顾名思义，模板数据库就是创建新 database 时所使用的骨架。创建新 database 时，PostgreSQL 会基于模板数据库制作一份副本，其中会包含所有的数据库设置和数据文件。

PostgreSQL 安装好以后默认附带两个模板数据库：`template0` 和 `template1`。如果创建新库时未指定使用哪个模板，那么系统默认会使用 `template1` 库作为新库的模板。



切记，任何时候都不要对 `template0` 模板数据库做任何修改，因为这是原始的干净模板，如果其他模板数据库被搞坏了，基于这个数据库做一个副本就可以了。如果你想定制自己的模板数据库，那么请基于 `template1` 进行修改，或者自己另外创建一个模板数据库再修改。对基于 `template1` 或你自建的模板数据库创建出来的数据库来说，你不能修改其字符集编码和排序规则。如果你想这么做，那么请基于 `template0` 模板来创建新数据库。

基于某个模板来创建新数据库的基本语法如下。

```
CREATE DATABASE my_db TEMPLATE my_template_db;
```

你可以使用任何一个现存的 database 作为创建新数据库时的模板。当需要对一个现存的 database 进行复制时，该功能特别有用。此外，你还可以将某个现存的数据库标记为模板数据库，对于这种被标记为模板的数据库，PostgreSQL 会禁止对其进行编辑或者删除。任何一个具备 CREATEDB 权限的角色都可以使用这种模板数据库。以超级用户身份运行以下 SQL 可使任何数据库成为模板数据库。

```
UPDATE pg_database SET datistemplate = TRUE WHERE datname = 'mydb';
```

如果你想修改或者删除被标记为模板的数据库，请先将上述语句中的 datistemplate 字段值改为 FALSE，这样就可以放开编辑限制。如果你还想将此数据库作为模板的话，修改完后记得将此字段值改回来。

2.4.2 schema的使用

schema 可以对 database 中的对象进行逻辑分组管理。如果你的 database 中有很多表，那么管理起来会很麻烦，可以考虑把它们分门别类放到不同 schema 中来进行管理。同一个 schema 中的对象名不允许重复，但同一个 database 的不同 schema 中的对象是可以重名的。如果你将数据库中所有表都塞到默认的 public schema 中（创建数据库时默认创建的 schema），迟早会遇到对象重名的问题。你可以自行决定如何管理和组织 schema。例如，假设要为一家航空公司设计 IT 系统，那么可以将飞机信息表及其日常维护信息表放到一个叫作 plane 的 schema 中，把所有机组人员及其人事信息放到名为 employees 的 schema 中，并把所有乘客相关信息放入名为 passengers 的 schema 中，这样就把所有信息分门别类隔离开了。

另外一种常见的管理 schema 的方法是基于角色的管理。当系统拥有多个客户端并且每个客户端的数据必须完全隔离时，这种方法特别合适。

假设你开了一家“宠物狗美容店”（俗称狗狗 SPA）。刚开始你用一张放在 public schema 中的名为 dogs 的表来存储所有宠物狗的信息。然后你的两个好朋友成了你的客户。最近政府颁布了一项新的隐私保护条例，要求客户数据必须完全隔离，即禁止一个客户看到另一个客户的狗狗的任何信息。为了达到这个要求，你可以为每个客户都建立一个单独的 schema，每个 schema 中建立同样的一张 dogs 表。创建 schema 的语句如下：

```
CREATE SCHEMA customer1;
CREATE SCHEMA customer2;
```

然后你把这些狗狗的数据从单一的 dogs 表分拆到不同客户的 schema 的 dogs 表中。最后为每个 schema 创建一个与之同名的可登录角色。所有狗狗的信息现在完全被分散到它们对应的 schema 下。当用户通过客户端登录到你的数据库进行美容预约操作时，他们就只能看到他们各自狗狗的相关数据。

别急，到这儿还没完，后面还有更妙的用法。我们之前让角色和 schema 同名，这样就可以用上另外一种很有用的技巧，在介绍这个技巧之前需要先介绍一下 search_path 这个系统变量。

前面已经说过，同一个 schema 中的对象是不允许重名的，但不同 schema 中的对象可以重名。例如，在所有 12 个 schema 中都有一张同名的 dogs 表。那么问题来了，当执行类似 SELECT * FROM dogs 这种语句时，PostgreSQL 是怎么知道要查的是哪个 schema 中的表呢？这个问题最简单的解决办法是在表名前加上所属 schema 的名称，比如 SELECT * FROM customer1.dogs；另一种方法是通过设置 search_path 变量来解决，比如可以设定为 customer1, public。当执行查询语句时，规划器会先到 customer1 schema 中查找 dogs，找不到再到 public schema 中寻找，再找不到就结束。

PostgreSQL 有一个罕为人知的系统变量叫作 user，它代表了当前登录用户的名字。执行 SELECT user 就能看到其值。系统变量 user 和 current_user 完全相同，用哪个都一样。

别忘了我们前面将 schema 的名称取得和登录用户名一致，现在可以充分利用这一点了。接下来在 postgresql.conf 中将 search_path 变量设成下面这样。

```
search_path = "$user", public;
```

好了，如果当前登录的角色是 customer1，那么所有的查询都会优先去 customer1 schema 中寻找目标表，如果找不到才会去 public schema 下找。最重要的一点是，这样我们系统中的 SQL 语句就只需要一种写法，而不用在每个客户的 SQL 中加上对应的 schema 名。这样就算你的客户数增长到几千个甚至是几十万个也没关系，系统中所有的 SQL 语句都不需要修改。一些需要所有人共享的公共表可以放到 public schema 中。

我们强烈推荐为每一个扩展包创建一个单独的 schema 来容纳其对象（参见 2.6.1 节）。安装一个新的扩展包时，会在数据库服务器上创建大量的表、函数、数据类型以及其他对象。默认情况下它们都会被安装到 public schema 中，这样日积月累之后 public schema 里面会被搞得一团糟。例如，完整的 PostGIS 扩展包安装后会创建数千个函数，如果你此前已经在 public schema 中创建了一些自己的表和函数，可以想象一下，在加进来这几千个表和函数后，要从中找到属于你自己的那些是多么痛苦的一件事情！

在安装扩展包之前，先为其创建一个 schema，语句如下：

```
CREATE SCHEMA my_extensions;
```

然后把这个新的 schema 加入 search_path：

```
ALTER DATABASE mydb SET search_path='$user', public, my_extensions;
```

安装扩展包时，记得在 CREATE EXTENSION 语句中将你为其创建的新 schema 声明为其归属 schema。



对于现有连接来说，ALTER DATABASE .. SET search_path 命令执行后是不能直接生效的，你需要断开此连接并重连才可以。

2.5　权限管理

PostgreSQL 的权限管理机制非常灵活而自由，因此要想管理得当是需要一些技巧的。PostgreSQL 的权限控制甚至可以精确到字段和记录级别。是的，你没看错，如有必要甚至可以为同一张表中每行记录的每个字段单独设置其访问权限。



行级权限控制（RLS，row level security）是 PostgreSQL 9.5 中首次引入的。该特性在所有的 PostgreSQL 服务实例中都可使用，但如果 PostgreSQL 是运行在开启了安全增强模式的 Linux（SELinux）上，还能打开一些更高级的安全控制特性。

要想完整地介绍所有关于权限管理的知识可能需要好几章的篇幅，因此本节仅介绍正常使用所必备的知识，同时会指导你避开一些隐蔽的“雷区”，这些“雷”一旦踩到，会导致你根本无法访问想要访问的内容，或者服务器上的数据得不到有效防护。

做好 PostgreSQL 的权限管理可不是件很轻松的事。利用 pgAdmin 工具的图形化界面来进行操作会简单一些，或者说至少能让你比较清楚地了解到系统当前权限设置的全貌。通过 pgAdmin 可以完成绝大多数权限管理工作。如果你得负责权限管理工作而你又是个 PostgreSQL 新手，那么建议使用 pgAdmin。如果等不了我们按部就班地慢慢介绍，你也可以直接跳到 4.2.4 节去学习。

2.5.1　权限的类型

PostgreSQL 有几十种权限，其中的一些基本不会用到。常见的几种权限包括 `SELECT`、`INSERT`、`UPDATE`、`ALTER`、`EXECUTE` 以及 `TRUNCATE`。

大多数权限需要上下文，也就是需要绑定一个特定的数据库对象才有意义。例如，一个角色拥有 `ALTER` 权限，却不指明在哪个数据库对象上拥有此权限，这是没有意义的。在 table1 上拥有 `ALTER` 权限，在 table2 上拥有 `SELECT` 权限，在 function1 上拥有 `EXECUTE` 权限，诸如此类的权限才有意义。另外，并不是所有权限都适用于所有数据库对象，比如一张表上的 `EXECUTE` 权限就完全说不通。

有些权限无须绑定数据库对象，比如 `CREATEDB` 和 `CREATE ROLE` 就是这样。



PostgreSQL 官方使用 privilege 来表示“权限”，其他数据库中“权限”一词可能用 right 或者 permission 来表达。

2.5.2　入门介绍

假设你已安装好 PostgreSQL，创建好了一个超级用户角色并设定好了密码。请参照以下步骤来创建其他角色并设定其权限。

(1) PostgreSQL 在安装阶段会默认创建一个超级用户角色以及一个 database，二者的名称都是 postgres。请以 postgres 身份登录服务器。
(2) 在创建你自己的首个 database 之前，需要先创建一个角色作为此 database 的所有者，所有者可以登录该库。语法如下：

```
CREATE ROLE mydb_admin LOGIN PASSWORD 'something';
```

(3) 创建 database 并设定其所有者：

```
CREATE DATABASE mydb WITH owner = mydb_admin;
```

(4) 然后用 mydb_admin 身份登录并创建 schema 和表。

2.5.3 GRANT

GRANT 命令是授予权限的基本手段。基本用法如下。

```
GRANT some_privilege TO some_role;
```

请牢记以下几条关于 GRANT 的使用原则。

- 很显然，只有权限的拥有者才能将权限授予别人，并且拥有者自身还得有 GRANT 操作的权限。这一点是不言而喻的，因为自己没有的东西当然给不了别人。
- 有些权限只有对象的所有者才能拥有，任何情况下都不能授予别人。这类权限包括 DROP 和 ALTER。
- 对象的所有者天然拥有此对象的所有权限，不需要再次授予。然而请特别注意：一个对象的所有者并不天然拥有此对象的子对象。例如，虽然你是某个 database 的属主，但这并不意味着你必然是该 database 下所有 schema 的属主。
- 授权时可以加上 WITH GRANT 子句，这意味着被授权者可以将得到的权限再次授予别人，从而实现权限传递。示例如下。

```
GRANT ALL ON ALL TABLES IN SCHEMA public TO mydb_admin WITH GRANT OPTION;
```

- 如果希望一次性将某一类对象的所有权限都授予某人，可以使用 ALL 关键字来指代所有对象，而不需要针对每一个对象都操作一遍。请注意，此处 ALL TABLES 涵盖了所有的普通表、外部表以及视图。

```
GRANT SELECT, REFERENCES, TRIGGER ON
ALL TABLES IN SCHEMA my_schema TO
PUBLIC;
```

- 如果希望将权限授予所有人，可以用 PUBLIC 关键字来指代所有角色。

```
GRANT USAGE ON SCHEMA my_schema TO PUBLIC;
```

官方手册的“GRANT”一节中对 GRANT 命令有极其详尽的说明，强烈推荐你先认真阅读一下此节，以免不小心设错权限导致系统安全隐患。

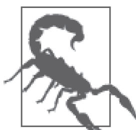
系统会默认将某些权限授予 PUBLIC（即所有人）。这些权限包括：CONNECT、CREATE TEMP TABLE（针对 database）、EXECUTE（针对函数）。有些情况下出于安全考虑，你可能希望取消

一些默认权限，那么可以使用 REVOKE 命令：

```
REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA my_schema FROM PUBLIC;
```

2.5.4 默认权限

默认权限可以简化权限管理工作，该机制允许用户在数据库对象创建之前就对其设置权限。



新增或者修改默认权限并不会影响已有的权限设置，即只有当某个对象的某项权限未专门设定的情况下，默认权限设定才会生效。

假设我们希望对所有数据库用户都授予某 schema 中将来可能创建的所有函数和表的 EXECUTE 和 SELECT 权限，那么我们可以按示例 2-8 这样来定义权限。一个 PostgreSQL 服务实例中的所有角色都是 PUBLIC 组的成员。

示例 2-8 定义 schema 的默认权限

```
GRANT USAGE ON SCHEMA my_schema TO PUBLIC; ❶

ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema
GRANT SELECT, REFERENCES ON TABLES TO PUBLIC; ❷

ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema
GRANT ALL ON TABLES TO mydb_admin WITH GRANT OPTION; ❸

ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema ❹
GRANT SELECT, UPDATE ON SEQUENCES TO public;

ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema ❺
GRANT ALL ON FUNCTIONS TO mydb_admin WITH GRANT OPTION;

ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema ❻
GRANT USAGE ON TYPES TO PUBLIC;
```

❶ 允许所有能够连接到此 database 的用户在 my_schema 中访问和创建对象，同时该用户需要已经具备访问此 schema 中所有对象的权限。为用户授予某个 schema 的 USAGE 权限是允许该用户访问 schema 中所有对象的前提条件。如果用户拥有访问 schema 中某张表的查询权限，却没有该 schema 的 USAGE 权限，则该用户不能访问这张表。

❷ 为所有具备此 schema 的 USAGE 权限的用户授予该 schema 中后续创建的所有表的查询（SELECT）和引用（REFERENCE）权限（引用权限指的是针对该表的某些字段建立外键约束的权限）。

❸ 把该 schema 中所有后续创建的表的所有权限都授予 mydb_admin 角色。同时还允许 mydb_admin 组的所有成员将本 schema 中所有后续创建的表的部分或者全部权限授予其他用户。所有权限具体包括：插入记录、更新记录、删除记录、截断表、创建触发器、创建约束等。

❹❺❻ 针对本 schema 中后续创建的序列号生成器、函数、数据类型授予权限。

要了解更多关于默认权限的信息，请参考官方手册中“修改默认权限”一节的内容。

2.5.5 PostgreSQL权限体系中一些与众不同的特点

最后，在你自己去深入了解权限管理体系之前，我们会给你列举一些比较隐蔽的“奇葩”特性。

与其他数据库不同，PostgreSQL 中一个 database 的所有者并不天然对此库中的所有对象拥有完全的控制权。比如另一个角色可以在你的库中创建一张表，你虽然身为此库的所有者却无权访问这张表，然而此时你却有权把整个库都删掉。

在对 schema 中的表和函数做了授权操作后，一定不要忘了授予 schema 本身的 `USAGE` 权限。

2.6 扩展包机制

扩展包（extension）是一种用于扩展 PostgreSQL 系统功能的插件机制，该机制的前身被称为“contrib”[3]。该机制很好地体现了开源界的强大优势：人们互相协作、共同开发并自由分享新的功能特性。自从 9.1 版开始引入对 extension 扩展包机制以来，为 PostgreSQL 添加功能扩展已经变得非常方便快捷。



对于在 extension 扩展包机制推出之前就已存在的那些历史插件，理论上我们应称之为“contrib”以示差别，但放眼未来，这些老的 contrib 势必都会被改造为用 extension 机制实现。为了描述方便，下文会将二者统称为“扩展包”，但你应该清楚二者的区别。

对于一台 PostgreSQL 服务器来说，并不是其中每个 database 都要安装全部的扩展包，只有当某个 database 的确需要此扩展包提供的功能时才应安装。如果你的 PostgreSQL 服务器上的所有 database 都需要某些扩展包的功能，那么可以新建一个模板数据库（有关模板数据库的介绍，请参见 2.4.1 节），然后在此模板数据库中预先安装好这些扩展包，那么后续的 database 就能以此模板数据库为基础来创建，这样就避免了每新建一个 database 就安装一遍扩展包的麻烦。

建议定期检查并卸载不再需要的扩展包以避免系统过于臃肿。将不再需要的老扩展包保留在系统中可能会在 PostgreSQL 升级过程中引发问题，因为升级后的 PostgreSQL 中需要把这些扩展包所含有的对象原地重建一遍。

要想查看某个 database 中已经安装了哪些扩展包，可以连接到该 database，然后执行示例 2-9 中的语句。你的查询结果可能和下面列出的很不一样，这是正常的。

示例 2-9　查询某个 database 中已安装的扩展

```
SELECT name, default_version, installed_version, left(comment,30) As comment
FROM pg_available_extensions
WHERE installed_version IS NOT NULL
ORDER BY name;
```

注 3：extension 与 contrib 本质上都是 PostgreSQL 的系统功能插件，二者功能类似但实现机制有很大差异，其版本分界点为 9.1 版，之前的插件机制被称为 contrib，9.1 版及之后的插件机制被称为 extension。
——译者注

```
name           | default_version | installed_version | comment
---------------+-----------------+-------------------+-------------------------------
btree_gist     | 1.5             | 1.5               | support for indexing common da
fuzzystrmatch  | 1.1             | 1.1               | determine similarities and dis
hstore         | 1.4             | 1.4               | data type for storing sets of
ogr_fdw        | 1.0             | 1.0               | foreign-data wrapper for GIS d
pgrouting      | 2.4.1           | 2.4.1             | pgRouting Extension
plpgsql        | 1.0             | 1.0               | PL/pgSQL procedural language
plv8           | 1.4.10          | 1.4.10            | PL/JavaScript (v8) trusted pro
postgis        | 2.4.0dev        | 2.4.0dev          | PostGIS geometry, geography, a
(8 rows)
```

如果你想查看一个 PostgreSQL 服务实例中所有 database 安装的所有扩展包，请把上面查询语句中的 `WHERE installed_version IS NOT NULL` 删掉。

如果想要了解某个 database 中已安装的扩展包的更多详细内容，请在 psql 中执行类似以下的命令：

```
\dx+ fuzzystrmatch
```

或者执行以下查询也可以：

```
SELECT pg_describe_object(D.classid,D.objid,0) AS description
FROM pg_catalog.pg_depend AS D INNER JOIN pg_catalog.pg_extension AS E
ON D.refobjid = E.oid
WHERE
D.refclassid = 'pg_catalog.pg_extension'::pg_catalog.regclass AND
deptype = 'e' AND
E.extname = 'fuzzystrmatch';
```

查询结果显示了该扩展包中包含了哪些内容：

```
description
----------------------------------------------------------------------------
function dmetaphone_alt(text)
function dmetaphone(text)
function difference(text,text)
function text_soundex(text)
function soundex(text)
function metaphone(text,integer)
function levenshtein_less_equal(text,text,integer,integer,integer,integer)
function levenshtein_less_equal(text,text,integer)
function levenshtein(text,text,integer,integer,integer)
function levenshtein(text,text)
```

扩展包中可以包含各类数据库对象，包括函数、表、数据类型、数据类型转换器、编程语言、运算符，等等。但函数通常占了其中的大部分。

2.6.1 扩展包的安装

将扩展包安装到系统中需要两个步骤：首先，下载安装包并安装到数据库服务器上；其次，将此扩展包安装到目标 database 中。



我们在前述两个步骤中都使用了“安装”这个词，但其指代的具体动作是不一样的，当上下文环境不清楚时，我们会加以描述区分。

以下我们将介绍扩展包的安装方法，同时也将介绍在不支持 extension 扩展包机制的老版本 PostgreSQL 上安装 contrib 扩展包的方法。

1. 步骤一：将扩展包安装到数据库服务器

这一步的具体做法会根据操作系统的不同而有所不同。总的来说就是先下载该扩展包的安装文件以及所依赖的库文件，然后将它们分别复制到操作系统的 bin 和 lib 文件夹，同时把 SQL 脚本文件复制到 share/extension 文件夹（9.1 版及之后版本）或者 share/contrib 文件夹（9.1 版之前的版本）。这样就为接下来执行第二步做好了准备。

对于较小的扩展包来说，其所需的很多库文件在 PostgreSQL 安装好以后就有了，或者没有的话也可以通过 `yum` 或 `apt get postgresql-contrib` 命令较容易地获取到。对于通过以上方式获取不到的库文件，你要么自行编译，要么找一下别人已经编译好的安装包，要么从另一台环境完全相同的服务器上把库文件复制过来。对于 PostGIS 这类较大的扩展包，通常可以从你下载 PostgreSQL 的站点下载到完整的安装包。如果想了解当前服务器上有哪些扩展包可用，请执行以下命令：

```
SELECT * FROM pg_available_extensions;
```

2. 步骤二：将扩展包安装到指定的database中

扩展包机制使得扩展特性的安装过程更加简单。使用 `CREATE EXTENSION` 命令即可将扩展包安装到指定的 database 中。相比原来的安装方法，该新机制有三大主要优点：首先，用户不需要弄清楚扩展包文件存放的具体路径（share/extension）；其次，可以通过 `DROP EXTENSION` 命令方便地卸载扩展包；最后，支持查看当前已安装和可安装的扩展包列表。PostgreSQL 安装包中已经附带了最常用的若干扩展包。通过 PostgreSQL Extension Network 站点可以下载到 PostgreSQL 安装包中默认未附带的扩展包。GitHub 站点上也有很多 PostgreSQL 扩展包，只需搜索“postgresql extension”关键字就能找到。

以下是安装 `fuzzystrmatch` 扩展包的命令：

```
CREATE EXTENSION fuzzystrmatch;
```

你仍可以使用 psql 以非交互方式安装扩展包。先连接到需要安装此扩展包的 database，然后执行类似以下命令行：

```
psql -p 5432 -d mydb -c "CREATE EXTENSION fuzzystrmatch;"
```



基于 C 语言的扩展包必须由具备超级用户权限的角色来安装。大多数扩展包都是基于 C 语言的。

强烈建议你创建专门的 schema 来安装扩展包，以确保扩展包数据与业务数据隔离。建好

schema 后，执行以下命令来将其指定给待安装的扩展包：

```
CREATE EXTENSION fuzzystrmatch SCHEMA my_extensions;
```

3. 升级PostgreSQL版本以支持新的extension扩展包机制

如果你从 PostgreSQL 9.1 或者更早的版本升级到了 9.1 版或者之后的版本，并且也正确地将数据从老版本导入到了新版本之中，那么之前安装的那些 contrib 扩展包依然是可以正常工作的。但为了简化管理并提升可维护性，你应该将存放于 contrib 文件夹中的那些老扩展包升级为新格式的 extension 扩展包。这种格式升级是完全可以实现的，尤其是对那些随 PostgreSQL 版本附带的扩展包来说更加没问题。不过请注意，这里说的"升级"是指从 contrib 格式到 extension 格式的"格式升级"，而不是指扩展包本身的"功能升级"。

例如，如果你之前在一套 PostgreSQL 9.0 版的 `contrib` schema 中安装了 `tablefunc` 扩展包（这是一个用于实现跨表查询的功能插件），然后你将该系统升级到了 9.1 版，那么可以执行以下命令来升级此插件：

```
CREATE EXTENSION tablefunc SCHEMA contrib FROM unpackaged;
```

该命令会搜索 `contrib` schema（假设你已把所有的扩展包都安装在这里面），找到所有属于 `tablefunc` 插件的成员对象，然后把它们按照新的 extension 扩展包模型格式进行封装，这样该扩展包就会出现在 `pg_available_extensions` 视图信息中，就好像是全新安装的一个 extension 扩展包一样。这样就实现了从 contrib 到 extension 的扩展包格式升级。

该命令会将 `contrib` schema 中的插件函数按 extension 模式进行封装格式升级，函数本身不会有任何改动，但此后该 database 进行备份时不会包含这些函数，因为它们的身份已经发生了变化，从与别的函数身份无法区分的普通函数变为 extension 扩展包中的函数。

2.6.2　通用扩展包

通用扩展包是指那些因功能比较基本和通用而在 PostgreSQL 安装包中默认附带的一些扩展包，但它们不一定会被默认安装，具体视其功能而定。有些早期的通用扩展包已被 PostgreSQL 内核接纳从而"登堂入室"成为系统基础功能，因此如果你是从较老版本的 PostgreSQL 升级上来的话，可能会发现原先要通过安装扩展包才能实现的功能现在已成了系统默认提供的功能。

1. 比较常用的扩展包介绍

从 9.1 版开始，PostgreSQL 官方推荐开发人员使用 extension 扩展包模式来为系统制作功能插件。从仅包含函数和数据类型的基本插件到包含了存储过程语言支持（PL）、索引类型以及外部数据封装器的高级插件，都可以用 extension 扩展包机制来实现。本节列举了最常用（也有些人称之为"必备"）但默认情况下 PostgreSQL 并未安装的一些扩展包。你会发现下面列出的很多扩展包在你的 PostgreSQL 系统中已经有了，具体哪些有哪些没有取决于你使用的是哪个 PostgreSQL 发行版，不同发行版之间可能会略有差异。

`btree_gist`

　该扩展包实现了基于 B-树索引算法的 GiST 索引运算符类，适用于 B-树索引支持的所有数据类型，其具体效果与标准的 B-树索引类似。更多细节请参见 6.3.1 节的内容。

`btree_gin`

该扩展包实现了基于 B-树索引算法的 GIN 索引运算符类，适用于 B-树索引支持的所有数据类型，其具体效果与标准的 B-树索引类似。更多细节请参见 6.3.1 节的内容。

`postgis`

该扩展包将 PostgreSQL 变成了一个业界最先进的空间数据库，而且其品质胜过了所有类似的商业化产品。如果你需要处理标准的 OGC GIS 数据、人口统计学数据、地理编码数据、3D 数据甚至是电子光栅数据，那么 `postgis` 会是你的必备之选。我们编写的图书 *PostGIS in Action* 中对 PostGIS 做了更加详尽的介绍。PostGIS 是扩展包界的“巨无霸”，它包含 800 多个函数、自定义数据类型以及空间索引等对象。PostGIS 如此庞大，以至于有人为它开发二次扩展包。有些二次扩展包是 PostGIS 安装包自带的，当然也有一些不是，其中有两个特别值得一提：一个是 `pgpointcloud`，它可以管理点云数据（地理信息系统中海量的点组成的集合），另一个是 pgRouting，它可以进行网络路径分析。

`fuzzystrmatch`

这是一个用于字符串模糊匹配的轻量级扩展包，包含了诸如 `soundex`、`levenshtein` 和 `metaphone` 等算法。我们在“Where is Soundex and Other Fuzzy Things”这篇文章中介绍了该扩展包的用法。

`hstore`

该扩展包为 PostgreSQL 添加了对键值数据库的支持，同时也支持索引，非常适用于存储非结构化数据。如果你正在寻找一种介于关系型数据库和 NoSQL 数据库之间的产品，可以尝试一下 `hstore`。很多原先适用 `hstore` 的场景已经可以用内置 `jsonb` 类型来替代，因此这个扩展包已经不像以前那么流行。

`pg_trgm`（trigram）

该扩展包提供了另外一种字符串模糊搜索算法库，可与 `fuzzystrmatch` 配合使用。该扩展包包含一种运算符类，使得基于 `ILIKE` 的搜索操作能用上索引。该扩展还能够让形如 `LIKE '%something%'` 的通配符查询或者是形如 `somefield ~ '(foo|bar)'` 的正则表达式查询用上索引。关于这部分内容，在“Teaching ILIKE and LIKE New Tricks”这篇博文中有更深入的探讨。

`dblink`

该扩展包支持从一台 PostgreSQL 服务器远程访问另一台 PostgreSQL 服务器上的数据。在 9.3 版中引入对外部数据源的支持之前，`dblink` 是唯一能够实现跨数据库交互的机制。目前该机制一般用于需要临时性连接到外部数据源或者临时的即席查询场景，特别是需要调用外部数据源一侧的函数时会有用。在 PostgreSQL 9.6 之前，`postgres_fdw` 不允许调用远端 PostgreSQL 服务器上除原生函数之外的函数，比如远端 PostgreSQL 安装了扩展包后所支持的新函数就不能调用。在 PostgreSQL 9.6 中，你可以预先声明远端服务器上已安装某扩展包，那么就可以在本地调用远端 PostgreSQL 服务器上安装的扩展包中的函数了。

`pgcrypto`

该扩展包提供了一系列的加密工具，包括使用广泛的 PGP 算法。使用该扩展包提供的

功能来对数据库中存储的信用卡号码或其他顶级机密信息进行加密是非常方便的。在“Encrypting Data with pgcrypto”这篇博文中我们对其用法进行了快速的入门介绍。

2. 经典扩展包介绍

此处我们介绍几个经典的“曾经的扩展包”。之所以称之为“曾经的扩展包”，是因为它们使用非常广泛，并因此被 PostgreSQL 官方接纳而成为了系统内核功能的一部分。但在老版本的 PostgreSQL 上，它们还是以扩展包的形式存在。你在实际工作中有可能遇到这些老版本，所以还是有必要介绍一下这些曾经的扩展包。

`tsearch`

　　该扩展包封装了一系列用于强化全文搜索功能的索引、运算符、自定义词典和函数。目前该扩展包的功能已被系统内核接纳并成为 PostgreSQL 基础功能的一部分。如果你使用的 PostgreSQL 版本较老，在其上 `tsearch` 还是以扩展包的形式存在，那么建议你升级到 `tsearch2` 这个新版本。当然，最好的做法其实是将 PostgreSQL 服务器软件升级到新版本，因为老版本上的 `tsearch` 插件的兼容性支持随时可能终止，这会导致你在老版本的 PostgreSQL 上再也无法享受到相关的功能更新和 bug 修复。

`xml`

　　该扩展包提供了对 XML 数据类型的支持以及相关的函数和运算符。为了达到 ANSI SQL XML 标准的要求，PostgreSQL 内核接纳了该插件包的部分功能。其余未被纳入内核的那部分功能仍以扩展包的形式存在，不过已改名为 `xml2`。具体来说，如果你需要使用 `xlst_process` 函数来处理 XSL 模板数据，那么就需要安装 `xml2` 扩展包。另外，`xml2` 扩展包中还含有一些 XPath 函数。

2.7　备份与恢复

PostgreSQL 自身附带了三个备份工具：`pg_dump`、`pg_dumpall` 和 `pg_basebackup`，三者均位于 bin 文件夹下。

`pg_dump` 用于备份一个指定的 database，而 `pg_dumpall` 可一次性备份所有 database 的数据以及系统全局数据。由于 `pg_dumpall` 需要能够访问系统中的所有 database，因此必须由具备 SUPERUSER 权限的角色来执行。`pg_basebackup` 可以针对所有 database 实现系统级的磁盘备份。

本节余下的内容将着重讨论 `pg_dump` 和 `pg_dumpall`。`pg_basebackup` 是实现整个 PostgreSQL 服务器备份的最有效工具。如果你的数据库足够大，比如 500GB 或者更大，那么你应该将 `pg_basebackup` 作为你备份策略中的一个重要工具。使用 `pg_basebackup` 时需要打开一些系统参数，这些参数一般是关闭的。另外，这些参数在配置复制功能时也要用到，因此我们把这部分留到 10.2 节再讨论。

这两个工具的大多数命令行选项都可以有两种写法：一种是 GNU 风格（两个横杠连字符后跟一个单词）；另一种是传统的单字母风格（一个横杠连字符后跟一个字母）。这两种写法是完全等价的，甚至在同一个命令行中也可以混用。本节仅讨论一些基本的用法，如果要了解更多深入的内容，请参考 PostgreSQL 官方手册中的“备份与恢复”。

业界也有一些常用的第三方工具来实现 PostgreSQL 的备份和恢复，但本节不会详细讨论。两个用得比较多的开源工具是 pgBackRest 和 Barman。相比官方原生的工具，它们多了定时备份、多服务器备份以及快捷恢复等功能。

在学习本节内容时，你会发现我们一般会在示例命令行中指明目标数据库所在的主机地址和侦听端口，这是因为我们一般是在另外一台机器上通过执行 pgAgent 定时任务的方式来执行备份，这种情况下必须在命令行写明目标数据库的地址和侦听端口，详情将在 4.5 节讨论；或者另外一种情况是在同一台机器上运行多个 PostgreSQL 服务实例，实例间的侦听端口互不相同，所以命令行中必须明确指定 IP 和端口。有一个情况请注意：如果你的服务被设置为仅侦听 localhost，会导致默认情况下只有来自本机的连接才会被允许接入，这样远端机器上的备份恢复工具会无法连接。如果备份任务是直接在数据库服务器本机上执行的，那么仅侦听 localhsot 是完全没问题的。

`pg_dump` 和 `pg_dumpall` 工具不支持在命令行选项中设定登录密码，因此为了便于执行自动任务，你需要在 `postgres` 操作系统账户的 home 文件夹下创建一个密码文件 .pgpass 来存储密码；或者也可以用 `PGPASSWORD` 环境变量来设定密码。

2.7.1 使用pg_dump进行有选择性的备份

如果你希望每天都进行备份，那么使用 `pg_dump` 比 `pg_dumpall` 更合适，因为前者支持精确指定要备份的表、schema 和 database，而后者不支持。`pg_dump` 可以将数据备份为 SQL 文本文件格式，也支持备份为压缩格式、TAR 包格式或者目录格式。在用 `pg_restore` 进行数据恢复时，前述三种格式的备份文件都可以实现并行恢复。用 `pg_dump` 进行备份时，选择目录格式可以实现并行备份，因此如果要备份的数据库非常大，可以考虑使用目录格式。

我们认为 `pg_dump` 是你进行日常备份时不可或缺的工具，因此我们在附录 B.1 节提供了一张完整的 `pg_dump` 帮助信息清单，这样你就可以对其数量众多的选项开关用法一目了然。

下面的例子展示了一些常见的备份场景以及相应的 `pg_dump` 选项。这些例子对于任何版本的 PostgreSQL 应该都是适用的。

备份某个 database，备份结果以自定义压缩格式输出：

```
pg_dump -h localhost -p 5432 -U someuser -F c -b -v -f mydb.backup mydb
```

备份某个 database，备份结果以 SQL 文本方式输出，命令中带 `-C` 选项，输出结果中包含 `CREATE DATABASE` 语句：

```
pg_dump -h localhost -p 5432 -U someuser -C -F p -b -v -f mydb.backup mydb
```

备份某个 database 中所有名称以“pay”开头的表，备份结果以压缩格式输出：

```
pg_dump -h localhost -p 5432 -U someuser -F c -b -v -t *.pay* -f pay.backup mydb
```

备份某个 database 中 `hr` 和 `payroll` 这两个 schema 中的所有数据，备份结果以压缩格式输出：

```
pg_dump -h localhost -p 5432 -U someuser -F c -b -v \
-n hr -n payroll -f hr.backup mydb
```

备份某个 database 中除了 public schema 中的数据以外的所有数据，备份结果以压缩格式输出：

```
pg_dump -h localhost -p 5432 -U someuser -F c -b -v -N public \
-f all_sch_except_pub.backup mydb
```

将数据备份为 SQL 文本文件，且生成的 INSERT 语句是带有字段名列表的标准格式，该文件可用于将数据导入到低于当前版本的 PostgreSQL 或者其他支持 SQL 的非 PostgreSQL 数据库中（之所以能够实现这种数据移植过程，是因为标准的 SQL 文本可在任何支持 SQL 标准的数据库中执行）：

```
pg_dump -h localhost -p 5432 -U someuser -F p --column-inserts \
-f select_tables.backup mydb
```



如果输出文件路径中含空格或者其他可能影响命令行正常处理的字符，请在路径两侧加上双引号，比如："/path with spaces/mydb.backup"。请注意，这在 PostgreSQL 中是一个通用的原则，即当你不确定某段文本是否能正常处理时，都可以加双引号。

从 PostgreSQL 9.1 版开始支持目录格式。使用该格式会将每个表备份为某个文件夹下的一个单独的文件，这样就解决了以其他备份格式备份时可能存在的单个文件大小超出操作系统限制的问题。pg_dump 工具所支持的各种备份格式中，只有使用目录格式时会生成多个文件，具体语法参见示例 2-10。备份时会先创建一个新目录，然后为每个表生成一个 gzip 格式的压缩文件，另外目录下还会生成一个描述所有备份对象之间层级关系的文件。如果备份开始时发现指定的目录已存在，那么该命令会报错并退出。

示例 2-10　目录格式备份

```
pg_dump -h localhost -p 5432 -U someuser -F d -f /somepath/a_directory mydb
```

从 9.3 版开始支持并行备份选项：--jobs 或 -j，后面设置并行度数值。例如将其设定为 --jobs=3（-j 3），则后台会有三个线程并行执行当前备份任务。此选项只有在按目录格式进行备份时才会生效，因为前面已经提到过，只有选择目录格式时才能并行生成多个备份文件。每个写线程只负责写一个单独的文件，因此一定是输出结果为多个独立的文件时才可以并行。示例 2-11 演示了其用法。

示例 2-11　目录格式并行备份

```
pg_dump -h localhost -p 5432 -U someuser -j 3 -Fd -f /somepath/a_directory mydb
```

2.7.2　使用pg_dumpall进行全局备份

pg_dumpall 工具可以将当前 PostgreSQL 服务实例中所有 database 的数据都导出为 SQL 文本（请注意：pg_dumpall 不支持导出 SQL 文本以外的其他格式）。可以看出，这是一种便于理解的明文备份格式，备份结果也包含了表空间定义和角色等全局对象。要了解该命令支持的所有选项，请参考附录 B.2 节。

建议你每天都对角色和表空间定义等全局对象进行备份，但不建议每天都使用 pg_dumpall

来备份所有 database 的数据，使用 `pg_dump` 来分别备份每个 database 或者使用 `pg_basebackup` 来备份整个 PostgreSQL 实例的全部数据是更好的选择。`pg_dumpall` 仅支持导出为 SQL 文本格式，而使用这种庞大的 SQL 文本备份来进行全库级别的数据恢复是极其耗时的。将 `pg_basebackup` 与流复制机制连用是实现 PostgreSQL 系统高可用的最快方法。

以下命令可实现仅备份角色和表空间定义等全局对象：

```
pg_dumpall -h localhost -U postgres --port=5432 -f myglobals.sql --globals-only
```

如果仅需备份角色定义而无须备份表空间，那么使用如下命令：

```
pg_dumpall -h localhost -U postgres --port=5432 -f myroles.sql --roles-only
```

2.7.3 数据恢复

PostgreSQL 可以使用以下两种方法来恢复 `pg_dump` 和 `pg_dumpall` 备份的数据：

- 使用 `psql` 来恢复 `pg_dump` 或者 `pg_dumpall` 工具生成的 SQL 文本格式的数据备份；
- 使用 `pg_restore` 工具来恢复由 `pg_dump` 工具生成的压缩格式、TAR 包格式或者目录格式备份。

1. 使用psql恢复SQL文本格式的数据备份

所谓的 SQL 文本格式的数据备份其实就是一个包含 SQL 脚本的文本文件。这种备份格式使用起来最不方便，却是最通用的一种。恢复时需要将此 SQL 脚本全部执行一遍。你无法选择性地仅恢复部分数据，除非你手工编辑此文件。以下示例都是在操作系统命令行界面执行，过程中可能需要在 psql 界面上输入密码。

恢复一个 SQL 备份文件并忽略过程中可能发生的所有错误：

```
psql -U postgres -f myglobals.sql
```

恢复一个 SQL 备份文件，如遇任何错误则立即停止恢复：

```
psql -U postgres --set ON_ERROR_STOP=on -f myglobals.sql
```

将 SQL 文本中的数据恢复到某个指定的 database：

```
psql -U postgres -d mydb -f select_objects.sql
```

2. 使用pg_restore进行恢复

如果你是使用 `pg_dump` 进行的备份，并且选择的输出格式为 TAR、压缩格式或者目录格式，输出结果都是二进制文件，那么必须使用 `pg_restore` 工具来进行恢复。`pg_restore` 支持的选项之多令人眼花缭乱，远远超过我们所使用过的其他任何数据库中提供的恢复工具。以下介绍一些该工具的特别功能。

- 支持并行恢复，使用 `-j`（或者是效果相同的 `--jobs=`）选项可以指定并行恢复的线程数。多个恢复线程可以并行处理，每个线程处理一张表。该模式可以显著提高恢复速度。
- 你可以使用 `pg_restore` 扫描备份文件来生成一张备份内容列表，通过该列表可以确认备份中包含了哪些内容。你还可以通过编辑该内容列表来控制恢复哪些内容。

- pg_restore 支持选择性地仅备份部分对象以节省备份时间，不管是针对整个 database 的备份还是针对部分对象的备份，都可以使用该特性。如果你仅需恢复单张表，那么可以用这个方法。
- pg_restore 的大部分功能是后向兼容的，即支持将老版本 PostgreSQL 生成的备份数据恢复到新版本的 PostgreSQL 中。

若想了解 pg_restore 命令所支持的全部选项，请参考附录 B.3 节。

在使用 pg_restore 执行恢复动作之前，请先创建目标数据库：

```
CREATE  DATABASE mydb;
```

然后执行恢复：

```
pg_restore --dbname=mydb --jobs=4 --verbose mydb.backup
```

如果备份和恢复时使用的 database 同名，则可以通过加 --create 选项省去单独建库的过程，命令如下：

```
pg_restore --dbname=postgres --create --jobs=4 --verbose mydb.backup
```

如果指定了 --create 选项，那么恢复出来的数据库名就会默认采用备份时的数据库名，不允许改名。如果还同时指定了 --dbname 选项，那么此时连接的数据库名一定不能是待恢复的数据库名，因为恢复数据库之前必然要建数据库，而建数据库之前必然要先连到某个已存在的数据库，--dbname 选项指定的就是建立被恢复的数据库之前先连到哪个数据库，所以必然不能与待恢复的数据库重名，我们一般指定为先连到 postgres 数据库。

正常情况下，数据恢复时不会重建目标 database 中已经存在的对象。如果你的目标 database 和其中的各类对象均已存在，并且还有一些数据在里面，你只是想用备份中的数据来替代现有 database 中的数据，那么执行 pg_resotre 时使用 --clean 选项即可达到目标。该选项的效果是先把目标 database 中的对象删除掉，然后在恢复过程中一一重建。



如果针对现存的一个 database 执行恢复，其内容会被备份文件中的内容替代。因此这种情况下需要特别注意：千万不要选错了备份文件或者指定了错误的目标 database。

9.2 版或更新版本的 pg_restore 支持 --section 选项，加上该选项后可以实现仅恢复表结构而不恢复表数据。当希望创建模板数据库时可以用这个方法，因为模板数据库一般不需要带数据。具体做法是先创建目标数据库：

```
CREATE DATABASE mydb2;
```

然后使用 pg_restore：

```
pg_restore --dbname=mydb2 --section=pre-data --jobs=4 mydb.backup
```

2.8 基于表空间机制进行存储管理

PostgreSQL 使用“表空间”这一概念来将逻辑存储空间映射到磁盘上的物理存储空间。

PostgreSQL 在安装阶段会自动生成两个表空间：一个是 `pg_default`，用于存储所有的用户级数据；另一个是 `pg_global`，用于存储所有的系统级数据。这两个表空间就位于默认的数据文件夹下。你可以不受限地创建表空间并将其物理存储位置设定到任何一块物理磁盘上。你也可以为 database 设定默认表空间，这样该 database 中创建的任何新对象都会存储到此表空间上。你也可以将现存的数据库对象迁移到新的表空间中。

2.8.1 表空间的创建

创建表空间需要先为其取一个逻辑名称并指定某个物理文件夹作为其存储位置，注意要确保 `postgres` 操作系统账户对此文件夹有完全的访问权限。如果你目前使用的是 Windows 服务器，请使用如下命令行（注意使用 Unix 风格的斜杠作为路径分隔符）：

```
CREATE TABLESPACE secondary LOCATION 'C:/pgdata94_secondary';
```

对于基于 Unix 的系统来说，必须先创建文件夹或者定义一个 fstab 位置，然后执行以下命令：

```
CREATE TABLESPACE secondary LOCATION '/usr/data/pgdata94_secondary';
```

2.8.2 在表空间之间迁移对象

你可以将数据库中的对象在表空间之间随意迁移。如果希望将一个 database 的所有对象都移动到另一个表空间中，可以执行以下命令：

```
ALTER DATABASE mydb SET TABLESPACE secondary;
```

如果只希望移动一张表，命令如下：

```
ALTER TABLE mytable SET TABLESPACE secondary;
```

PostgreSQL 9.4 中引入了一个新功能：一次性把一个表空间的多个对象迁移到另一个表空间。如果命令执行者是超级用户，那么源表空间所有的对象都会被迁移过去；否则仅会迁移执行者所拥有的对象。

将 `pg_default` 默认表空间中的所有对象迁移到 `secondary` 表空间，所需的命令行如下：

```
ALTER TABLESPACE pg_default MOVE ALL TO secondary;
```

在迁移过程中所涉及的 database 和表会被锁定。

2.9 禁止的行为

我们作为第一见证人见证并处理过很多 PostgreSQL 故障，因此在本章的最后一节中，我们认为有必要逐条列出那些最常见的错误。

对于初学者来说，如果你搞不清到底哪里出了问题，那么请首先查看系统日志，从中可以找到解决问题的线索。日志文件位于数据文件夹的根目录或者其中的 pg_log 子文件夹下。也有可能系统在记下日志之前即已崩溃，那么显然这种情况下查日志是没用的。如果你的 PostgreSQL 服务启动失败，请尝试执行以下操作系统命令。

```
path/to/your/bin/pg_ctl -D your_postgresql_data_folder
```

2.9.1 切记不要删除PostgreSQL系统文件

你可能觉得这是废话，但当磁盘空间不够的时候有些人就会慌手慌脚地从 PostgreSQL 的数据文件夹下删除文件，因为它占用的空间实在是太大了。出现这种问题的部分原因是有些文件夹的名称的确很容易令人误解，比如：pg_log、pg_xlog 和 pg_clog。这里面的确有些文件是可以安全删除的，但你需要准确地知道哪些能删哪些不能删，否则很容易导致数据库被破坏。

pg_log 文件夹一般在 data 文件夹下，其体积可能增长得很快，尤其是当打开了日志开关的时候。这个文件夹下的文件任何时候都可以安全删除，事实上很多人会设置一个定时任务来定期清除这些日志文件。

除了 pg_xlog 文件夹下的文件可以有条件地删除外，其他 PostgreSQL 系统文件夹中的文件都不能删，即使有的文件夹名称中带有 log 字样因而看起来像是某种日志，也绝对不能删，比如 pg_clog 文件夹中存储的是活跃事务提交日志，除非你想删库跑路，否则千万不要碰。

pg_xlog 文件夹用于存储事务日志。我们见过有的系统中会在 pg_xlog 文件夹下建一个子文件夹 archive，专门用于存放归档的事务日志。一般来说，你的系统总会需要创建一个专门的文件夹（该文件夹不一定要放在 pg_xlog 下）用于日志归档，因为如果这是一个同步复制环境，那么需要持续地进行事务日志归档；需要归档文件夹的另一个理由是得把这些日志存下来以备不时之需，因为我们有可能需要将系统数据恢复到过去的某个时间点。将 pg_xlog 文件夹下的所有文件都删除会导致 PostgreSQL 后台进程崩溃。但仅删除归档文件夹下的日志却没这么严重，最多会导致无法恢复到过去的某个时点，或者是在同步环境下可能导致从属服务器无法进行数据同步，因为还未来得及同步的一些日志文件可能已被删掉了。如果以上场景在你的系统中都不涉及，那么可以放心地删除归档文件夹下的日志文件。

要当心某些过于“尽责”的杀毒软件，特别是在 Windows 上。我们曾经遇到过杀毒软件把很重要的 PostgreSQL 可执行文件给删掉的案例。如果在 Windows 环境下发现 PostgreSQL 无法启动，记得首先查看一下事件查看器（event viewer）的记录，其中可能存在有用的线索。



在 PostgreSQL 10 中，pg_xlog 目录被改名为 pg_wal，pg_clog 被改名为 pg_xact，以防止用户认为这些目录中放的都是日志，想删就删而不会造成任何后果。

2.9.2 不要把操作系统管理员权限授予PostgreSQL的系统账号

很多人可能会误认为 `postgres` 这个操作系统账号必须拥有操作系统的管理员权限。事实上，在有的 PostgreSQL 版本上，如果给予了 `postgres` 账号管理员权限，很有可能导致该操作系统启动失败。

`postgres` 账号应该就是一个普通用户账号，只要能够访问 data 文件夹以及其他表空间文件夹即可。大多数 PostgreSQL 安装包会自动为 `postgres` 账号设定够用的权限，请不要画蛇添足。在 SQL 注入攻击中，那些不必要的权限会为攻击者们提供可乘之机。

有些情况下，的确是有必要把 data 文件夹以外的某些文件夹或者可执行程序的“改写 / 删除 / 读取”权限授予 postgres 账号。比如，当需要设置一个定时任务来执行批处理任务或者访问文件型 FDW 时就需要这么干。针对这种情况，我们也建议你仅仅授予 postgres 账号最小的必要权限，而不应该为了图省事而直接授予其最高权限。

2.9.3 不要把shared_buffers缓存区设置得过大

禁止将 shared_buffers 设置得和系统当前内存总容量一样大，否则很可能会导致操作系统崩溃或者是启动失败。在 32 位 Windows 系统上，该设置如果超过 512MB 就可能导致系统出问题。在 64 位 Windows 上，该限制可以放宽到 1GB 或者更大一些也没问题。在有些 Linux 系统上，不能将 shared_buffers 设置得大于 SHMMAX 变量，而一般来说 SHMMAX 的值是比较小的，所以这就给 shared_buffers 的设置带来了一些限制。

PostgreSQL 9.3 修改了内核对于内存的使用方法，因此之前版本中那些因内核限制而导致的问题从该版本开始不复存在。官方手册中“内核资源管理”一节介绍了更多关于这方面的细节。

2.9.4 不要将PostgreSQL服务器的侦听端口设为一个已被其他程序占用的端口

如果启动 PostgreSQL 时系统发现侦听端口已被占用，那么会在 pg_log 中记录类似这样的错误日志：make sure PostgreSQL is not already running。以下是可能导致该问题的常见原因。

- postgres 服务实例已经启动好了，而你正试图再启动一遍。
- PostgreSQL 服务侦听端口已被其他程序占用。
- postgres 服务之前发生过异常关闭或者崩溃，在 data 文件夹下遗留了一个 postgresql.pid 文件。请直接删除该文件并再次尝试启动。
- 有一个孤立的 PostgreSQL 进程还在运行。如果前述方法都试过了但问题仍未解决，请终止所有还在运行的 PostgreSQL 进程，然后再次尝试启动。

第 3 章

psql工具

psql 是 PostgreSQL 自带的一个不可或缺的命令行工具，用途广泛，除了执行 SQL 这个基本功能外，还可用于执行脚本、导入导出数据、恢复表数据以及执行其他数据库管理任务，它甚至还可以作为一个简单的报表生成器来使用。如果你只能以无图形界面的命令行方式访问数据库服务器，那么 psql 就是你访问 PostgreSQL 的唯一选择。如果你符合前述情况，那么你就得熟悉 psql 的众多命令和选项。在学习本章时，建议你将附录 B.4 节中提供的 psql 帮助信息打印出来，放在手边以备查阅。

3.1　环境变量

在设置 PGHOST、PGPORT 和 PGUSER 等环境变量后，在调用 psql 命令行时就不用显式地指定主机、端口和用户了，系统会自动使用环境变量设定的值，这一点跟 PostgreSQL 自带的其他命令行工具是一样的。为避免每次登录时都输入密码，你也可以用 PGPASSWORD 这个环境变量来设置登录密码。如果希望以更安全的方式处理登录密码，请使用密码文件，相关内容请参见官方手册中“密码文件”一节的介绍。从 PostgreSQL 9.2 版开始，psql 支持以下两个新的环境变量。

PSQL_HISTORY

　该变量用于设置 psql 历史日志文件名，该日志中记录了近期通过 psql 执行过的所有命令行，其默认值为 ~/.psql_history。

PSQLRC

　该变量用于设置用户自定义的配置文件的路径和文件名。你可以自行决定是否使用该特性。用户可以将常用的设置项统一集中存放到自定义配置文件里，然后 psql 启动时会先读取这个文件再加载默认配置，这个文件中的配置会覆盖默认配置。

如果你既未设定相应的环境变量，也未在命令行中指定相关参数，那么 psql 会使用系统默认值。



如果你使用 pgAdmin3 工具连接到了某个 database，那么可以通过其工具栏上的“插件”菜单项来直接打开 psql，这个新打开的 psql 使用的参数就是 pgAdmin 配置的参数。

3.2 psql的两种操作模式：交互模式与非交互模式

直接在操作系统的命令行界面上键入 `psql` 并按回车，从操作系统提示符切换到 psql 提示符后就表示已经进入了 psql 的交互模式界面。你现在就可以执行命令了。对于 SQL 语句来说，记得要输入分号作为命令结束标记，要是不输入就直接按回车的话，psql 会认为命令还未输入完成，会在换行后等待继续输入。

在 psql 界面上键入 `\?` 会列出交互模式下支持的所有命令。为了方便读者，我们将这些命令列表放在了附录 B 中，并高亮显示最新版本中添加的条目，具体请参见附录 B.4 节。如果在 psql 界面上键入 `\h`，后跟命令关键字，则会打印出该命令在 PostgreSQL 官方手册中相应的语法帮助信息。

如果希望顺序执行一系列命令，我们建议把这些命令写成一个脚本文件，然后使用 psql 以非交互模式执行该脚本。在操作系统的命令提示符下输入 `psql` 后带脚本文件名即可执行该脚本。脚本中可以含有任意数量的 SQL 和 psql 命令。除执行脚本外，非交互模式还支持直接执行一条或多条 SQL 语句，不过语句两侧需要加上双引号。可以看出，该模式特别适用于需要执行自动化任务的场景。只需将任务内容写入脚本文件，然后用某种定时任务工具来设置好定时执行即可。定时任务工具可以采用 pgAgent，在 Linux/Unix 下可以使用 `crontab`，在 Windows 下可以使用定时任务规划器。

由于非交互模式下绝大多数操作逻辑都由脚本实现，因此命令行需要提供的选项很少。如需了解非交互模式下 psql 的选项，请参考附录 B.5 节。要在非交互模式下执行脚本文件，只需使用 `-f` 选项即可：

```
psql -f some_script_file
```

要在非交互模式下执行 SQL 语句，只需使用 `-c` 选项即可。如果要一次执行多个语句，语句之间请用分号分隔：

```
psql -d postgresql_book -c "DROP TABLE IF EXISTS dross; CREATE SCHEMA staging;"
```

在脚本中也可以使用交互式命令。示例 3-1 是一个名为 build_stage.psql 的脚本的内容，其中会创建一张名为 `staging.factfinder_import` 的表，该表后续会在示例 3-10 中用到。脚本中先生成了一个 `CREATE TABLE` 命令，然后将这个命令写入了 create_script.sql 文件，最后执行了刚刚生成的 create_script.sql 文件。

示例 3-1　带 psql 交互式命令的脚本

```
\a ❶
\t
```

```
\g create_script.sql
SELECT
    'CREATE TABLE staging.factfinder_import (
        geo_id varchar(255), geo_id2 varchar(255), geo_display varchar(255),' ||
        array_to_string(array_agg('s' ||
        lpad(i::text,2,'0') || ' varchar(255),s' ||
        lpad(i::text,2,'0') || '_perc varchar(255)'),',') ||
    ');'
FROM generate_series(1,51) As i;
\o  ❷
\i create_script.sql  ❸
```

❶ 我们希望该脚本执行后输出的结果是能直接运行的 SQL 语句，因此需要用 `\t`（或者 `--tuples-only`）选项来忽略标题栏的输出，同时使用 `\a` 选项关闭对齐模式以防止 psql 为对齐输出结果而自动加上换行符。使用 `\g` 让所有查询内容都输出到指定文件。

❷ 使用不带参数的 `\o` 命令来停止查询结果重定向到外部文件。

❸ 使用 `\i` 加脚本名 create_script.sql 来执行生成的脚本。`\i` 的效果等同于非交互模式下的 `-f` 选项。

要执行示例 3-1，在操作系统命令行界面下输入以下命令行即可：

```
psql -f build_stage.psql -d postgresql_book
```

示例 3-1 中借鉴了“How to Create an N-column Table”这篇博文中所介绍的方法来创建表。这篇文章中介绍的方法无须借助中间文件，而是采用了从 PostgreSQL 9.0 版开始支持的 `DO` 命令来执行自动化建表。

3.3　定制psql操作环境

如果你的工作需要整天与 psql 打交道，那么可以对 psql 操作环境进行定制以使其更好地符合自己的使用习惯。psql 在启动阶段会搜索一个名为 psqlrc 的配置文件，如果找到则会顺序执行其中的配置动作，这些配置决定了 psql 的一些行为模式。

在 Linux/Unix 环境中，该文件一般会被命名为 .psqlrc 并放置在 postgres 用户的 home 目录下。在 Windows 上，该文件叫作 psqlrc.conf 并被放置于 %APPDATA%\postgresql 文件夹下，一般来说就是 c:\Users\username\AppData\Roaming\postgresql 文件夹。PostgreSQL 安装完成后找不到此文件是正常的，因为该文件一般需手动创建。该文件中的设置项会覆盖 psql 的默认值。如需了解更多关于此配置文件的信息，请参见官方手册中“psql 参考手册”一节的内容。

示例 3-2 中展示了 psqlrc 文件的内容，你可以在其中添加任何 psql 命令以在启动时执行。

示例 3-2　psqlrc 文件内容

```
\pset null 'NULL'
\encoding latin1
\set PROMPT1 '%n@%M:%>%x %/# '
\pset pager always
\timing on
```

```
\set qstats92 '
    SELECT usename, datname, left(query,100) || ''...'' As query
    FROM pg_stat_activity WHERE state != ''idle'' ;
'
```



psqlrc 文件中 set 命令后跟的操作内容不允许分为多行书写。上文显示为两行仅仅是因为该语句较长导致印刷时放不下，所以必须分成两行。

启动 psql 时，屏幕上会显示该文件中内容被执行后的输出结果。

```
Null display is "NULL".
Timing is on.
Pager is always used.
psql (9.6beta3)
Type "help" for help.
postgres@localhost:5442 postgresql_book#
```

有的 psql 设置命令仅适用于 Linux/Unix 环境而不适用于 Windows 环境，反之亦然。但不管在什么操作系统环境下，在指定路径时都应使用 Linux/Unix 风格的正斜杠（/），其目的是为了区别于指定选项时所用的反斜杠（\）。如果希望 psql 启动时跳过加载 psqlrc 文件这一步骤，使用默认参数，请添加 -X 选项。

你可以在 psql 运行时动态修改参数，但必须关闭 psql 再打开才能生效。如果要删除某个 psql 配置变量或者希望将其设置回默认值，可以使用 \unset 命令，后跟变量名称，比如：\unset qstat92。

请注意：使用 set 命令时设置的变量名是区分大小写的。系统变量请使用大写，用户自定义变量请使用小写。在示例 3-2 中，PROMPT1 是一个系统变量，用于指定 psql 终端界面上的提示符，而 qstat92 是一个自定义变量，作用是提供一个查询 PostgreSQL 服务器上当前活跃连接的快捷方式。

3.3.1 自定义psql界面提示符

如果你工作时需要使用 psql 在多台不同的数据库服务器或者多个不同的 database 间切换，那么定制不同的提示符是很有必要的。定制后的 psql 界面提示符可以告诉你当前连接的是哪台服务器上的哪个 database，从而避免误操作的发生。下面是设置带有很多信息的提示符的一种简单方式：

```
\set PROMPT1 '%n@%M:%>%x %/# '
```

其中包括了几个元素：登录角色（%n）、主机名（%M）、侦听端口（%>）、事务状态（%x）以及当前使用的 database 名（%/）。这种写法可能详细过头了，你可以按需裁剪一下。提示符所支持的完整符号列表可从官方手册的“psql 参考手册”一节中找到。

连接到数据库后，提示符看起来会像下面这样：

```
postgres@localhost:5442 postgresql_book#
```

执行 `\connect postgis_book` 切换目标数据库后，提示符会变成下面这样：

```
postgres@localhost:5442 postgis_book#
```

3.3.2 语句执行时间统计

有时候我们可能会需要 psql 打印出执行每个语句所消耗的时间。可通过 `\timing` 命令来打开或者关闭执行时间统计开关。

打开执行时间统计开关时，每个查询执行完毕的输出结果中都会附带执行时长。例如，使用 `\timing on` 命令执行 `SELECT COUNT(*) FROM pg_tables` 后，输出如下：

```
count
--------
73
(1 row)
Time: 18.650 ms
```

3.3.3 事务自动提交

默认情况下，自动提交功能是处于启用状态的，也就是说任何一个 SQL 语句执行完毕后，它所做的数据修改都会被立即提交，这种情况下每个语句都是一个独立的事务，一旦执行完毕其结果就不可撤销。如果你需要运行大量的 DML 语句并且这些语句还未经充分测试，那么自动提交功能会带来大麻烦，此时有必要关闭事务自动提交机制来保护数据。

请先关闭自动提交功能：`\set AUTOCOMMIT off`。然后就可以按需对事务进行回滚了：

```
UPDATE census.facts SET short_name = 'This is a mistake.';
```

要回滚事务，请执行：

```
ROLLBACK;
```

要提交事务，请执行：

```
COMMIT;
```



在非自动提交模式下请一定要记得在最后提交事务，否则未提交的事务会在退出 psql 时自动回滚掉。

3.3.4 命令别名

你可以使用 `\set` 来为某个命令创建别名，我们建议你将全局性的别名写到 psqlrc 文件中。例如，如果你每十分钟就要执行一次 `EXPLAIN ANALYZE VERBOSE`，那么每次完全重输一遍会很烦人，可以为它起一个别名。

```
\set eav 'EXPLAIN ANALYZE VERBOSE'
```

这样在任何需要用到 EXPLAIN ANALYZE VERBOSE 命令的地方，都可键入 :eav 替代（前面的冒号表示这是一个需要展开的命令变量）。

```
:eav SELECT COUNT(*) FROM pg_tables;
```

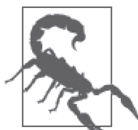
你甚至可以为一个完整的查询语句起个别名并存入 psqlrc 文件中，如同我们在示例 3-2 中所做的一样。建议别名使用小写字母，以避免与大写的系统环境变量冲突。

3.3.5 取出前面执行过的命令行

跟许多命令行工具一样，你在 psql 中也可以用向上方向键快速找出之前执行过的历史命令。HISTSIZE 环境变量决定了系统存储的历史命令行的数量。例如：\set HISTSIZE 10 会将可追溯的历史命令数量设定为最多 10 条。

如果你正在编写一个非常复杂的查询语句或者执行一系列很关键的更新操作，那么可能会需要将这些语句存入指定的文件中以备后续查看，可以使用 HISTFILE 环境变量来实现此功能。

```
\set HISTFILE ~/.psql_history - :DBNAME
```



Windows 环境下不支持保存历史命令，除非是使用 Cygwin、MingW 或者 MSYS 来模拟 Unix 环境。

3.4 psql使用技巧

本节将介绍一些被湮没在大量的 psql 文档中的特殊技巧。

3.4.1 执行shell命令

psql 中通过 \! 可以直接执行操作系统命令。比如你使用的是 Windows 环境，需要列出当前工作目录的内容，那么不需要退出 psql 环境，只需直接在 psql 界面上执行 \! dir 即可。

3.4.2 用watch命令重复执行语句

\watch 命令是 PostgreSQL 9.3 中为 psql 引入的一项新功能。它可以实现以固定的频率反复执行某个语句，以便持续观察其输出。例如你需要持续监控系统中当前正在执行的所有语句的情况，那么只需把 watch 语句加到查询语句的后面即可，如示例 3-3 所示。

示例 3-3　每 10 秒钟查询一次所有数据库连接上的活跃负载

```
SELECT datname, query
FROM pg_stat_activity
WHERE state = 'active' AND pid != pg_backend_pid();
\watch 10
```

虽然设计 \watch 命令的本意是用于反复执行监控类查询语句以便于持续观察系统状态，但你也可以将其用于重复执行其他指定的语句，watch 命令并不关心语句本身的内容是什么。

在示例 3-4 中，我们先使用批量加载方法创建了一张表，然后每 5 秒记录一次系统负载信息。注意后面只有跟着 \watch 的第二句话才会被每 5 秒执行一次。

示例 3-4　每 5 秒记录一次系统负载情况

```
SELECT * INTO log_activity
FROM pg_stat_activity; ❶
INSERT INTO log_activity
SELECT * FROM pg_stat_activity; \watch 5 ❷
```

❶ 以 pg_stat_activity 为模板新建一张结构和数据都完全相同的 log_activity 表。

❷ 每 5 秒重复一次将 pg_stat_activity 最新数据导入 log_activity 的动作。

如果需要终止 watch 进程，请执行 CTRL-X 加 CTRL-C。

3.4.3　显示对象信息

有多条 psql 命令都能用于显示数据库对象列表，并附带给出每个对象的详细信息。示例 3-5 展示了如何列出 pg_catalog 中以 pg_t 打头的所有表的信息，同时附带这些表所占空间的大小。

示例 3-5　使用 \dt+ 命令列出表信息

```
\dt+ pg_catalog.pg_t*

Schema     | Name              | Type  |  Owner   |  Size  | Description
-----------+-------------------+-------+----------+--------+-------------
pg_catalog | pg_tablespace     | table | postgres | 40 kB  |
pg_catalog | pg_trigger        | table | postgres | 16 kB  |
pg_catalog | pg_ts_config      | table | postgres | 40 kB  |
pg_catalog | pg_ts_config_map  | table | postgres | 48 kB  |
pg_catalog | pg_ts_dict        | table | postgres | 40 kB  |
pg_catalog | pg_ts_parser      | table | postgres | 40 kB  |
pg_catalog | pg_ts_template    | table | postgres | 40 kB  |
pg_catalog | pg_type           | table | postgres | 112 kB |
```

如果需要查询某个特定对象的详细信息，可以使用 \d+ 命令，如示例 3-6 所示。

示例 3-6　通过 \d+ 命令得到对象的详细信息

```
\d+ pg_ts_dict

Table "pg_catalog.pg_ts_dict"
Column         | Type | Modifiers | Storage  | Stats target | Description
---------------+------+-----------+----------+--------------+-------------
dictname       | name | not null  | plain    |              |
dictnamespace  | oid  | not null  | plain    |              |
dictowner      | oid  | not null  | plain    |              |
dicttemplate   | oid  | not null  | plain    |              |
dictinitoption | text |           | extended |
Indexes:
"pg_ts_dict_dictname_index" UNIQUE, btree (dictname, dictnamespace)
"pg_ts_dict_oid_index" UNIQUE, btree (oid)
Has OIDs: yes
```

3.4.4 行转列视图

PostgreSQL 9.6 中 psql 新增支持了 \crosstabview 命令，这一特性极大地降低了视图行转列的难度。该命令可以为用户节约很多时间，但只能在 psql 命令行环境下使用。示例 3-7 演示了如何使用这一特性，后面也会给出详细的说明。

示例 3-7　行转列视图

```
SELECT student, subject, AVG(score)::numeric(5,2) As avg_score
FROM test_scores
GROUP BY student, subject
ORDER BY student, subject
\crosstabview student subject avg_score

 student | algebra | calculus | chemistry | physics | scheme
---------+---------+----------+-----------+---------+--------
 alex    | 74.00   | 73.50    | 82.00     | 81.00   |
 leo     | 82.00   | 65.50    | 75.50     | 72.00   |
 regina  | 72.50   | 64.50    | 73.50     | 84.00   | 90.00
 sonia   | 76.50   | 67.50    | 84.00     | 72.00   |
(4 rows)
```

\crosstabview 命令应紧跟在你希望实现行转列的 SQL 语句后面。\crosstabview 命令的输入是前面 SQL 语句中查询的三个字段，后面还可以跟一个可选的排序字段。该命令实现的效果是对原生查询结果进行行转列后重排：第一个字段是一行的标记，第二个字段被翻转为列，第三个字段按照第一个和第二个字段的每种排列组合作为值出现。执行 \crosstabview 命令时也可以不输入字段名，这就要求前面的 SQL 查询语句中只能查询三个字段，命令默认就会去取这三个字段作为入参。

在示例 3-7 中，student 是行的标记字段，subject 是行转列后铺开的字段，avg_score 会在前两个字段排列组合而成的每个交叉点处作为值出现。如果查询结果中某个特定的 student-subject 组合不存在，则其值为空。我们可以在 \crosstabview 命令的入参中显式输入字段名，但当 SQL 本身的查询字段列表就是我们想要的列表时，我们可以不输入。

3.4.5 执行动态SQL

有时我们会需要根据某个查询的结果动态构造出 SQL 语句，然后执行它。在 PostgreSQL 9.6 之前的版本中，需要先构造出 SQL，然后将其输出到外部脚本文件，再执行脚本文件。当然，你也可以选择使用 DO 语法，但对于长 SQL 语句来说用起来很不方便。从 PostgreSQL 9.6 开始，你可以选择使用 \gexec 命令来执行动态生成的 SQL，其特点是生成 SQL 和执行 SQL 在同一步骤中完成，非常方便。该命令会遍历查询输出结果中的每一行并执行其中的 SQL 语句。遍历顺序是外层按记录行迭代，内层按字段迭代。gexec 命令还没有支持识别每行记录中的每个字段是否都是一个 SQL，目前仅支持每行是一个 SQL 的情况。另外，gexec 仅仅机械地遍历执行所有 SQL，并不关心每个 SQL 的执行结果，即使中途有 SQL 执行出错，gexec 依然会继续遍历执行下去，但它在遍历查找 SQL 的过程中会识别出 null 记录并跳过。示例 3-8 中创建了两张表并使用 \gexec 实现了往每张表中插入一条记录。

示例 3-8　使用 gexec 创建表并插入数据

```
SELECT
    'CREATE TABLE ' || person.name || '( a integer, b integer)' As create,
    'INSERT INTO ' || person.name || ' VALUES(1,2) ' AS insert
 FROM (VALUES ('leo'),('regina')) AS person (name) \gexec

CREATE TABLE
INSERT 0 1
CREATE TABLE
INSERT 0 1
```

在示例 3-9 中，我们使用 gexec 来查询 information_schema 中的表，获取元数据。

示例 3-9　使用 gexec 获取每张表中的记录数

```
SELECT
'SELECT ' || quote_literal(table_name) || ' AS table_name,
COUNT(*) As count FROM ' || quote_ident(table_name) AS cnt_q
FROM information_schema.tables
WHERE table_name IN ('leo','regina') \gexec

table_name | count
-----------+------
leo        | 1
(1 row)

table_name | count
-----------+------
regina     | 1
(1 row)
```

3.5　使用psql实现数据的导入和导出

psql 支持一个叫作 \copy 的命令，该命令可以将数据导出到文本文件中，也可以从文本文件中导入数据。文本文件中默认使用制表符作为分隔符，当然你也可以指定使用其他分隔符。文本中必须使用换行符来分隔不同的行，否则无法正确区分两行记录。我们采用美国人口普查数据网站上提供的马萨诸塞州人口统计数据来作为我们的第一个例子。你可以从链接 http://www.postgresonline.com/downloads/postgresql_book_2e.zip 中下载我们接下来要用到的 DEC_10_SF1_QTH1_with_ann.csv 文件。

3.5.1　使用psql进行数据导入

需要导入非规范化数据或者需要导入我们不太了解其特征的数据时，我们一般建议这么做：首先，新建一个独立的 schema 来作为数据过渡区，将新数据导入此 schema 中；然后，通过一些查询来摸清这些数据的特性；最后才把这些数据分门别类导入到正式的产品表中，并删除之前建立的过渡区 schema。

在将数据导入 PostgreSQL 之前，需要先建立一张表来容纳新数据，而且表的列数和数据类型必须与待导入的数据一致。如果一个待导入的文本文件的内部列和记录结构清晰有序，那么这个建表步骤理论上是可以省略的，因为 psql 可以自动判断每个列的类型并自动

把表建好，但为了避免出现 psql 自动判断数据类型出错的情况，这个步骤最好不要省略。psql 把整个导入过程当成一个完整的事务来处理，因此如果导入数据时遇到任何错误，那么整个导入动作所做的修改会完全回滚掉。如果你对源文件中数据的特点并不完全了解，我们建议你用最宽松的条件来创建容纳表，等数据导入之后再对数据进行细化加工。例如，如果你不确定某一列是否全都是数字，那么可以将该列的数据类型设置为 `character varying`，等数据导入之后再执行检查，稍后再进行转换。

示例 3-10 会向我们在示例 3-1 中创建的表中导入数据。打开 psql 并执行示例 3-10 中的命令。

示例 3-10　使用 psql 导入数据

```
\connect postgresql_book
\cd /postgresql_book/ch03
\copy staging.factfinder_import FROM DEC_10_SF1_QTH1_with_ann.csv CSV
```

在上面的示例中，我们在 psql 的交互模式下执行了导入过程：首先连接数据库，然后使用 `\cd` 切换到含有数据源文件的目录，最后执行 `\copy` 导入动作。因为 `\copy` 命令支持的默认分隔符是制表符，所以我们必须在命令行中额外指明源文件是用逗号作为分隔符的 CSV 格式。

如果源文件使用了一些非标准的分隔符，比如竖杠（|），那么也请在命令中以如下方式指明：

```
\copy sometable FROM somefile.txt DELIMITER '|';
```

如果希望把文本中的空值替换为你指定的内容再导入，可以用 `NULL AS` 来标记要替换的内容：

```
\copy sometable FROM somefile.txt NULL As '';
```



请不要将 psql 中的 `\copy` 命令与 SQL 语言提供的 `COPY` 语句相混淆。psql 是一个客户端工具，所有路径都是相对于已连接客户端进行解释的。而 SQL copy 是基于服务器的，并在 `postgres` 服务操作系统账户的环境下运行。因此输入文件必须驻留在可由 `postgres` 服务账户访问的路径中。

3.5.2　使用psql进行数据导出

psql 的数据导出功能比导入功能更简单、更灵活，你甚至可以指定仅导出某张表的某一部分记录。导出时使用的依然是 psql 的 `\copy` 命令。在示例 3-11 中，我们将演示如何将前面刚刚导入库中的数据再导回到以制表符为分隔符的文本中。

示例 3-11　使用 psql 导出数据

```
\connect postgresql_book
\copy (SELECT * FROM staging.factfinder_import  WHERE s01 ~ E'^[0-9]+' )
TO '/test.tab'
WITH DELIMITER E'\t' CSV HEADER
```

默认情况下 psql 导出数据时会使用 tab 键作为分隔符。然而，以这种格式导出时默认不导出标题行。你可以通过指定 `HEADER` 选项来要求导出标题行，请注意该选项仅当输出格式为 `CSV` 时才可使用（参见示例 3-12）。

示例 3-12　使用 psql 导出数据

```
\connect postgresql_book
\copy staging.factfinder_import TO '/test.csv'
WITH CSV HEADER QUOTE '"' FORCE QUOTE *
```

FORCE QUOTE * 表示输出的所有列的前后都将加上引用符，这个引用符默认就是双引号，但为清晰起见，我们还是显式指定了一下。

3.5.3　从外部程序复制数据以及将数据复制到外部程序

从 PostgreSQL 9.3 版开始，psql 开始支持从命令行程序的输出中获取数据并将数据转储到表中，这类命令行程序包括 curl、ls 和 wget 等。示例 3-13 演示了如何使用 dir 命令将一个目录下的文件列表导入表中。

示例 3-13　使用 psql 导入某个目录下的文件列表

```
\connect postgresql_book
CREATE TABLE dir_list (filename text);
\copy dir_list FROM PROGRAM 'dir C:\projects /b'
```

Hubert Lubaczewski 在博文“Piping copy to/from an external program”中介绍了更多关于使用 \copy 命令的例子。

3.6　使用psql制作简单的报表

你也许会觉得难以置信，但 psql 的确能够制作简单的 HTML 报表。请执行以下命令并查看生成的 HTML 报表，它应该类似图 3-1 所展示的样子。

```
psql -d postgresql_book -H -c"
SELECT category, count(*) As num_per_cat
FROM pg_settings
WHERE category LIKE '%Query%'
GROUP BY category
ORDER BY category;
" -o test.html
```

category	num_per_cat
Query Tuning / Genetic Query Optimizer	7
Query Tuning / Other Planner Options	5
Query Tuning / Planner Cost Constants	6
Query Tuning / Planner Method Configuration	11
Statistics / Query and Index Statistics Collector	6

(5 rows)

图 3-1：简单的 HTML 报表

上面的报表看起来还算凑合，但这仅仅是输出了一个 HTML 表格，还称不上是一个完全符合格式要求的 HTML 文档。为了让该报表的内容更丰富一些，我们需要写一个脚本来作为辅助，内容见示例 3-14。

示例 3-14　编写 settings_report.psql 文件来设置报表内容

```
\o settings_report.html ❶
\T 'cellspacing=0 cellpadding=0' ❷
\qecho '<html><head><style>H2{color:maroon}</style>' ❸
\qecho '<title>PostgreSQL Settings</title></head><body>'
\qecho '<table><tr valign=''top''><td><h2>Planner Settings</h2>'
\x on ❹
\t on ❺
\pset format html ❻
SELECT category,
string_agg(name || '=' || setting, E'\n' ORDER BY name) As settings ❼
FROM pg_settings
WHERE category LIKE '%Planner%'
GROUP BY category
ORDER BY category;
\H
\qecho '</td><td><h2>File Locations</h2>'
\x off ❽
\t on
\pset format html
SELECT name, setting FROM pg_settings WHERE category = 'File Locations'
ORDER BY name;
\qecho '<h2>Memory Settings</h2>'
SELECT name, setting, unit FROM pg_settings WHERE category ILIKE '%memory%'
ORDER BY name;
\qecho '</td></tr></table>'
\qecho '</body></html>'
\o
```

❶ 指定查询结果输出到一个文件中。

❷ HTML 表格的输出格式设置。

❸ 添加一些附加的 HTML 代码。

❹ 打开记录输出的展开模式。重复每一个记录的列标题，并将每一个记录的每一列作为一个单独的记录输出。

❺ 强制查询的结果输出为一个 HTML 表格。

❻ `string_agg()` 是 PostgreSQL 9.0 中引入的一个函数，可以将聚合运算中被划为同组的字符串值合并为单个字符串。

❼ 关闭记录输出的展开模式，这样第二个和第三个查询结果在报表上的输出格式应该是每条记录仅占一行。

❽ 设置“是否仅输出记录”开关。如果此开关是开启的，则会忽略列标题和行计数。

示例 3-14 演示了通过灵活使用 SQL 和 psql 命令可以创建出一个内容丰富的综合性分层报表。要运行示例 3-14 中的脚本有两种方法：可以使用 psql 以交互方式连接并执行 `\i settings_report.psql`，也可以在操作系统的命令行界面上运行 `psql -f settings_report.psql`。settings_report.html 生成的输出结果如图 3-2 所示。

Planner Settings

category	Query Tuning / Other Planner Options
settings	constraint_exclusion=partition cursor_tuple_fraction=0.1 default_statistics_target=100 from_collapse_limit=8 join_collapse_limit=8

category	Query Tuning / Planner Cost Constants
settings	cpu_index_tuple_cost=0.005 cpu_operator_cost=0.0025 cpu_tuple_cost=0.01 effective_cache_size=16384 random_page_cost=4 seq_page_cost=1

category	Query Tuning / Planner Method Configuration
settings	enable_bitmapscan=on enable_hashagg=on enable_hashjoin=on enable_indexonlyscan=on enable_indexscan=on

File Locations

config_file	C:/projects/pg/pg92edb/data/postgresql.conf
data_directory	C:/projects/pg/pg92edb/data
external_pid_file	
hba_file	C:/projects/pg/pg92edb/data/pg_hba.conf
ident_file	C:/projects/pg/pg92edb/data/pg_ident.conf

Memory Settings

maintenance_work_mem	16384	kB
max_prepared_transactions	0	
max_stack_depth	2048	kB
shared_buffers	4096	8kB
temp_buffers	1024	8kB
track_activity_query_size	1024	
work_mem	1024	kB

图 3-2：复杂的 HTML 报表

如上所示，通过构造 psql 脚本可以实现将多个查询的输出结果整合到一个报表中。另外，你还可以通过 pgAgent、crontab 或者 Windows 定时任务管理器等工具定时执行脚本。

第4章

pgAdmin的使用

pgAdmin4 是一款 PostgreSQL 图形化管理工具，其可靠性和实用性已久经考验，目前最新版本是 V1.6。相较其前任 pgAdmin3，pgAdmin4 经过了彻底的改写。pgAdmin3 所支持的一些特性还没有移植到 pgAdmin4，但将来都会移植过来。本章将聚焦于 pgAdmin4 的现有功能。pgAdmin4 的大部分功能在 pgAdmin3 中也都有，因此本章内容对于 pgAdmin3 用户也有价值。本章还会介绍一些在 pgAdmin3 中运用广泛但尚未移植到 pgAdmin4 的特性。后续我们将统一使用 pgAdmin 来称呼这两个版本，仅当二者在功能上有差异时才会加上版本号。



到目前为止，pgAdmin4 相对于 pgAdmin3 的主要变化包括：对于较新的 PostgreSQL 9.6 和 PostgreSQL 10 支持较好；支持以浏览器或桌面应用两种模式运行；支持在查询结果窗格中直接对记录进行编辑；支持同时选中查询结果窗格中的非邻接项；性能方面的提升。如果你在 Windows 平台上工作，请一定要使用 pgAdmin4 的 V1.6 或者更高的版本，V1.6 之前的版本在 Windows 下以桌面模式运行时存在性能问题。

虽然 pgAdmin 有其缺点，但开发组对 bug 的修复一直都很及时，而且也在不停地为其添加新的功能特性。因为 PostgreSQL 社区的开发者们将其定位为使用最广泛的 PostgreSQL 图形化管理工具，并且在很多 PostgreSQL 发行包中都会附带，所以开发者们一直确保它与最新版本的 PostgreSQL 保持同步更新。如果新版本的 PostgreSQL 中引入了一个新特性，那么最新版本的 pgAdmin 一定会支持对此特性进行管理。如果你是 PostgreSQL 新手，那么选择 pgAdmin 作为入门工具肯定没错。

4.1 pgAdmin入门

很多 PostgreSQL 发行包中都附带有 pgAdmin4。BigSQL 和 EDB 的 PostgreSQL 发行包自

从 9.6 版之后就包含了 pgAdmin4 作为一个可选安装项。如果你需要使用 pgAdmin3 工具来管理 PostgreSQL 9.6 之后的版本，可以使用 BigSQL 提供的 pgAdmin3 LTS，该版本已经针对 PostgreSQL 9.6 和 PostgreSQL 10 做了适配更新。用 BigSQL 公司的包管理器工具可以安装 pgAdmin3 LTS。从 PostgreSQL 9.5 起，EDB 公司的发行包中就只包含 pgAdmin4 了。pgAdmin 开发组也不会再对 pgAdmin3 做任何的功能增强。

如果只需安装 pgAdmin 而无须安装 PostgreSQL 本身，你可以从 pgAdmin 的官网下载 pgAdmin 的安装包。该网站还提供了 pgAdmin 的使用手册，你可以仔细研读一下。该工具的功能设计清晰有序，而且大部分功能是自解释的，也就是说你仅凭摸索而无须专门指导即可了解其使用方法。对于那些爱"尝鲜"的用户来说，可以尝试使用测试版，PostgreSQL 社区将很感谢你对测试版进行试用并反馈 bug。

4.1.1 功能概览

下面会先列出我们认为最精华的部分功能，不过这只是小试牛刀，更多内容请参见 pgAdmin 官网上的功能介绍页面。

同时支持服务器模式和桌面应用模式

pgAdmin4 同时支持以桌面模式和服务器模式安装，前者是以本地应用的方式运行，后者是一个基于 WSGI 规范的 Web 应用，可以通过浏览器访问。pgAdmin3 仅支持桌面模式。

执行计划的图形化解释功能

该功能非常有用，它能够以图形化方式展示执行计划。当然原来那种冗长的文本形式的执行计划也会继续存在，但图形化方式展示的执行计划会让用户对整个执行计划了解得更加深入和全面。

SQL 执行面板

不管在界面上执行什么样的操作，pgAdmin 最终都要通过 SQL 语句来与 PostgreSQL 服务端进行交互，系统允许你查看这种底层 SQL。当你使用图形化界面对数据库服务器进行操作时，pgAdmin 会自动在 SQL 结果窗格中展示这些自动生成的 SQL 语句。对于新手来说，研究这种自动生成的 SQL 是极好的学习途径。对专家来说，好好利用这种自动生成的 SQL 可以节省大量时间。

postgresql.conf 和 pg_hba.conf 等配置文件的图形化编辑器

你不再需要四处寻找配置文件的位置并使用文本编辑器来修改它们，在 pgAdmin 上可以一站式搞定。该功能当前仅在 pgAdmin3 中支持，而且使用该功能需要预先在 `postgres` database 中安装名为 `pgadmin` 的扩展包。

数据导入和导出

pgAdmin 能够轻易地将语句查询结果导出为 CSV 文件或者基于其他分隔符的文本文件，当然也可以将这类文件导入到数据库中。pgAdmin3 甚至支持将表数据导出为 HTML 格式，因此可以当作一个简易的一键式报表服务器来用。

备份与恢复向导

如果你记不住 `pg_restore` 和 `pg_dump` 命令的大量选项，没关系，pgAdmin 提供了一个

很友好的图形化界面来帮你设定这些选项，通过调整这些选项可以实现对 database、schema、单张表以及全局对象进行定制化的备份或者恢复。你可以在备份恢复界面的"消息"选项卡上看到系统自动生成的 `pg_dump` 或 `pg_restore` 命令行语句，如果你觉得有必要，完全可以把这些语句复制下来作为例句使用。

授权向导

该功能可以帮助你一次性对很多数据库对象进行授权或者解除授权操作，从而大大节省你的时间。

pgScript 脚本执行引擎

pgScript 是一种速度很快但不太"正规"的脚本执行机制，该机制不要求整个脚本中执行的所有操作构成一个事务。在 pgScript 中，你可以在一个循环语句的每一次循环中都执行一次提交操作，如果是在函数中，那么只能是所有循环都结束后才能执行提交。很遗憾的是，这种灵活的机制只能在 pgAdmin3 中使用（pgAdmin4 还不支持）。

SQL 编辑器的自动补全功能

用 CTRL 加空格键可以激活自动补全功能，该功能在 pgAdmin4 中得到了进一步的完善。

pgAgent 定时任务工具

pgAgent 是一种跨平台的定时任务计划工具，后续将使用一整节来介绍它。pgAdmin 提供了一套很完善的用于访问 pgAgent 的接口。

4.1.2 如何连接到PostgreSQL服务器

通过 pgAdmin 连接到 PostgreSQL 服务器很简单，其通用（General）和连接（Connection）选项卡如图 4-1 所示。

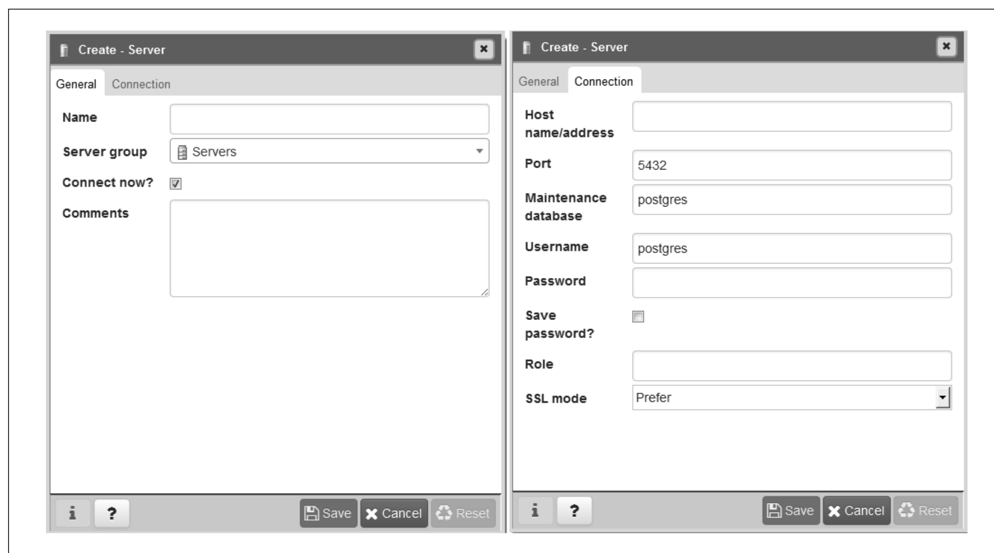


图 4-1：pgAdmin4 注册服务器连接对话框

4.1.3 pgAdmin界面导航

pgAdmin 界面左侧的树状目录布局看起来很直观，但由于它直接把所有底层数据库对象呈现给了用户，所以可能让人毫无头绪。你可以打开 Files → Preferences 页面，点开左侧的 Nodes 页面，然后勾选掉你不希望看到的数据库对象类型，这样主页面上显示的目录树就会精简很多。可以通过菜单栏上的 Files（文件）→ Preferences（偏好）→ Browser（浏览器）→ Nodes（显示节点）来打开目录树定制面板。你将看到如图 4-2 所示的界面。

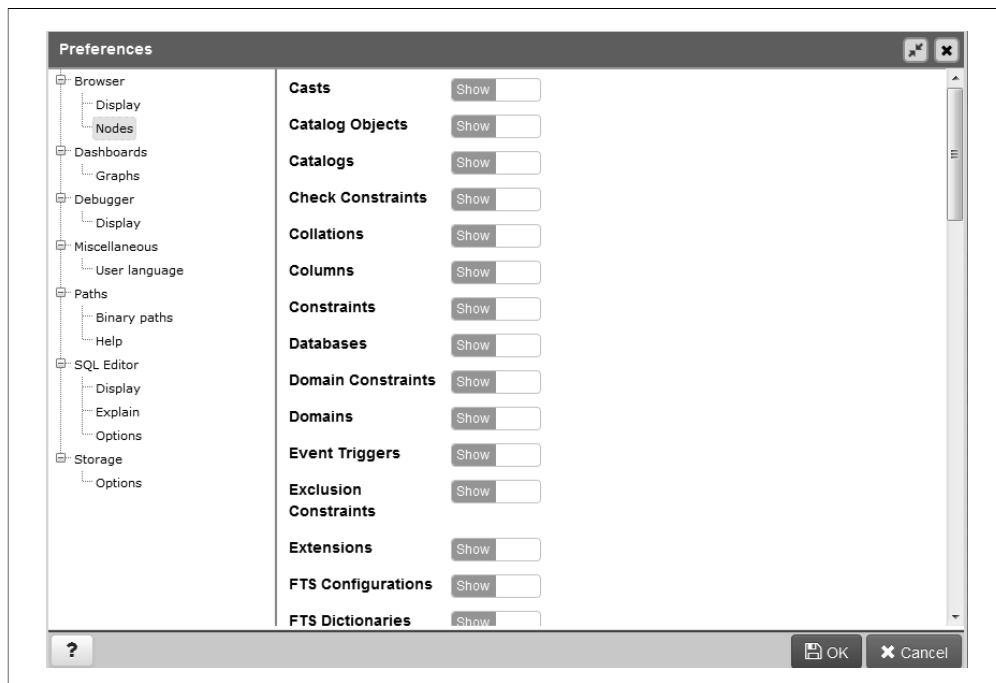


图 4-2：在 pgAdmin4 的树状浏览目录中隐藏或者显示特定类型的数据库对象

如果在 Browser → Display 页面上勾选了“在树状目录中显示系统对象”，那么你将看到 PostgreSQL 服务器的内部对象，包括内部函数、系统表、表的隐藏字段等。你还将看到 PostgreSQL 系统 schema 中存储的元数据，包括 `information_schema` 和 `pg_catalog` 这两个 catalog 中的内容。其中 `information_schema` 是 ANSI SQL 标准中规定必须要有的，因此在别的数据库（比如 MySQL 和 SQL Server）中也会存在。你可能会认出一些在使用其他数据库产品时曾经见过的表和字段。

4.2 pgAdmin功能特性介绍

pgAdmin 工具的功能丰富而强大，本书的篇幅不足以全面描述，因此我们仅重点介绍一些比较常用的功能。

4.2.1 根据表定义自动生成SQL语句

pgAdmin 有一个基于表定义自动生成 SELECT、INSERT、UPDATE 语句模板的菜单。在表名上点击右键，滚动到其中的 Scripts 栏，即可找到该菜单，界面如图 4-3 所示。

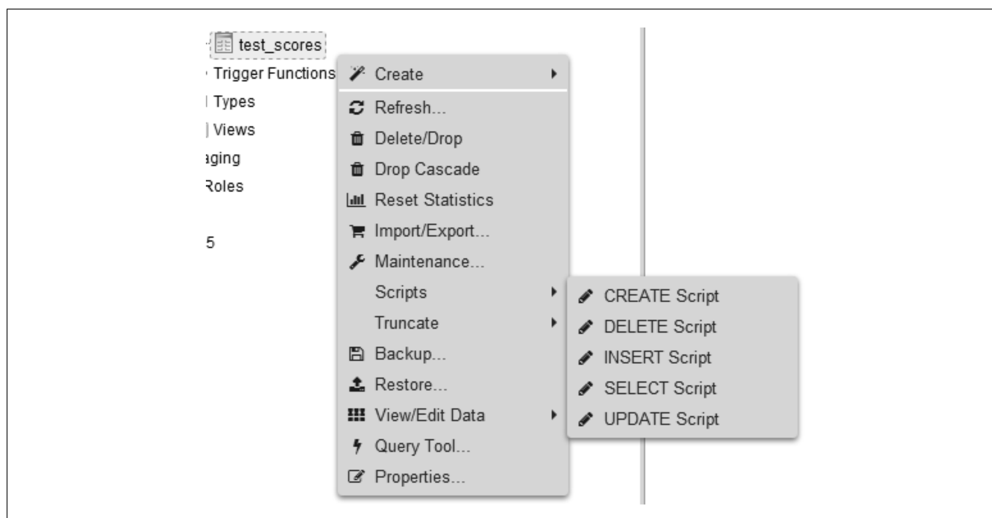


图 4-3：表脚本自动生成菜单

其中的“SELECT Script”项特别好用，它一点开就会自动生成一个包含了该表所有字段的查询语句。如果表有很多字段，而你又需要查询其中的大部分字段，那么这个自动生成的语句会为你节省很多时间，只需把你不想要的字段从脚本中删除即可。

4.2.2 在pgAdmin3中调用psql

尽管 pgAdmin 拥有功能强大的图形界面，但有些时候还是离不开 psql。比如需要执行 pg_dump 或其他数据转储工具生成的体积庞大的 SQL 文件时，psql 就更合适。从 pgAdmin3 界面很容易打开 psql 工具（请注意该特性仅在 pgAdmin3 中可用，pgAdmin4 还不支持），只需点击“插件”菜单下的“PSQL Console”项即可，如图 4-4 所示。这样就会启动一个 psql 窗口并连接到当前 pgAdmin 环境中已连接的 database 上，然后你可以使用 \cd 以及 \i 命令来改变目录并执行 SQL 文件。

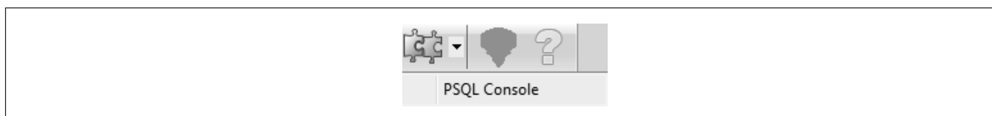


图 4-4：psql 插件

请注意：该功能需要确保针对某 database 的连接已建立好，因此只有当 pgAdmin 已连上 PostgreSQL 服务器并选中某个 database 时，“插件”菜单下的“PSQL Console”项才会变成可用状态。

4.2.3 在pgAdmin3中编辑postgresql.conf和pg_hba.conf文件

只要服务器上安装了 `adminpack` 扩展包，那么你就可以在 pgAdmin 界面上直接编辑配置文件。一般来说，PostgreSQL 的一键式安装包都会自动安装好 `adminpack` 扩展包。只要该插件已安装，你就可以看到 Server Configuration（服务器配置）菜单已启用，如图 4-5 所示。

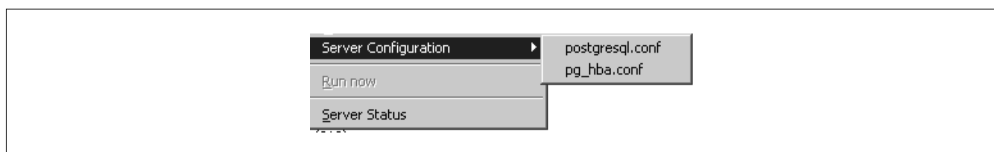


图 4-5：pgAdmin3 配置文件编辑器

如果你的 pgAdmin 已连接到 PostgreSQL 服务器但 Server Configuration 菜单却是灰的，那么要么是没安装 `adminpack`，要么是你不是以超级用户身份登录的。如果要安装 `adminpack`，执行 `CREATE EXTENSION adminpack` 即可，或者也可以通过图形界面来安装，如图 4-6 所示。安装好以后请断开与服务器的连接并重连，然后就可以看到菜单已可点击。

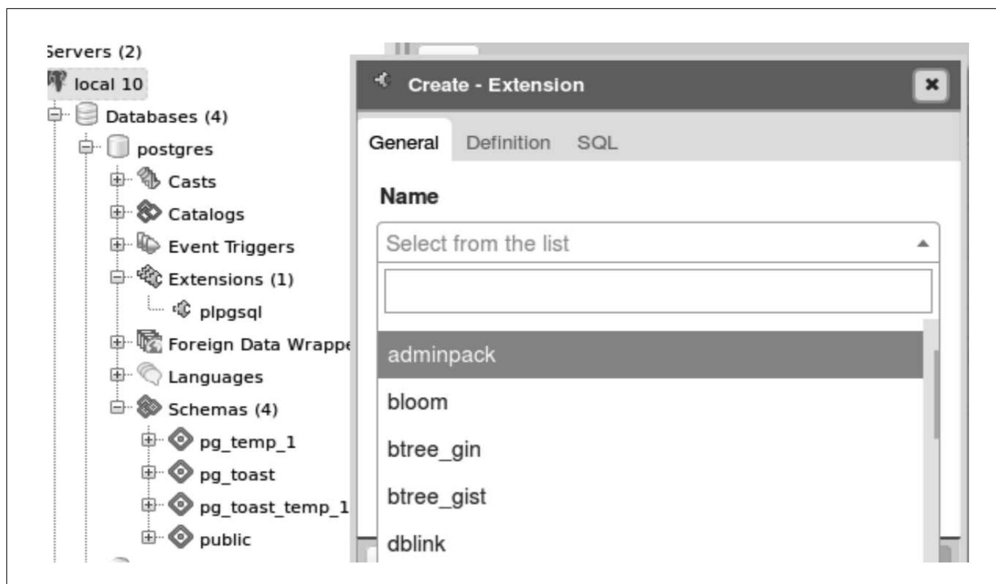


图 4-6：使用 pgAdmin4 安装扩展包

4.2.4 创建数据库对象并设置权限

pgAdmin 允许你创建各种数据库对象并对其进行权限设置。

1. 创建数据库以及其他数据库对象

利用 pgAdmin 创建一个新的数据库非常简单，只需右键单击树上的 database 节点并选择 New Database（新建数据库）即可，如图 4-7 所示。Definition（定义）选项卡上提供了一个下拉菜单供选择建库所用的模板数据库，我们在 2.4.1 节中介绍过相关内容。

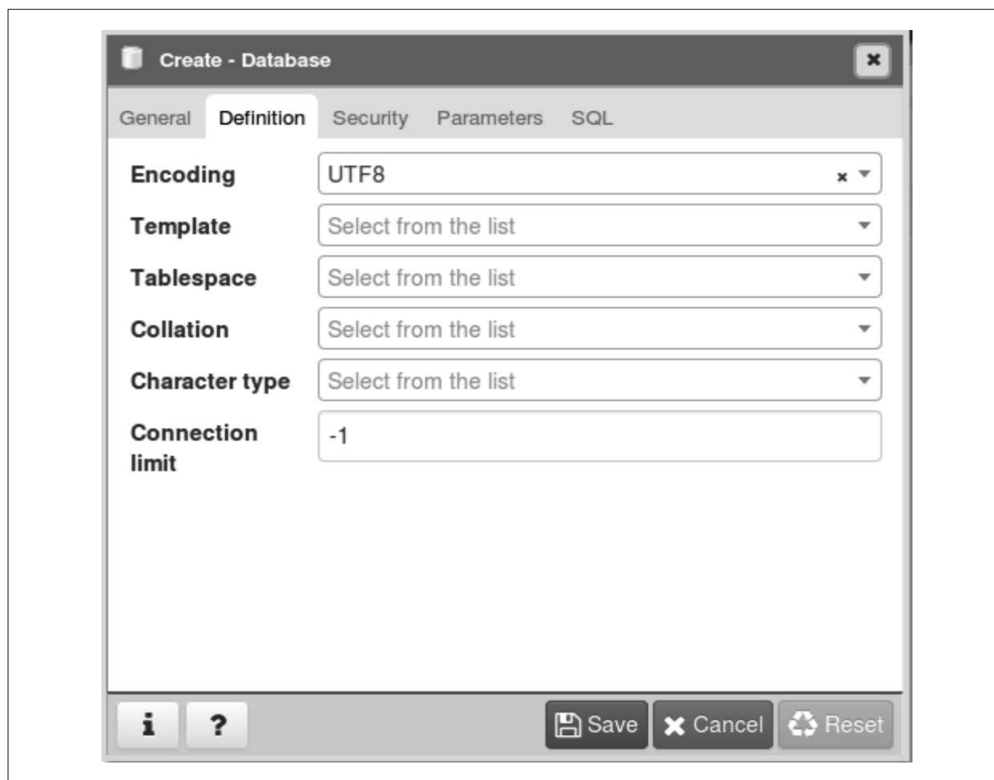


图 4-7：使用 pgAdmin4 创建新数据库

创建角色、schema 和其他数据库对象的步骤是类似的，都有一些对应的相关页面供你设置其他属性。

2. 权限管理

在 PostgreSQL 数据库对象权限管理方面，不会有比 pgAdmin 的授权向导更好的管理工具了，你可以通过菜单栏上的 Tools（工具）→ Grant Wizard（授权向导）打开其页面。如果你只需要对 schema 中的对象进行授权，可以直接在 schema 名字上点击右键并选择 Grant Wizard，打开的页面上只会出现该 schema 所拥有的对象。如同其他许多功能项一样，在成功连到数据库之前，其菜单项一直都是灰的。另外，该菜单项对于当前树状目录上的焦点位置也很敏感，点击到不同位置时，该菜单项就会显示出可用或者不可用等不同状态。例如，要为 cesus 这个 schema 中的项设置权限，请在目录树上选中此 schema 并点开授权向导，界面如图 4-8 所示。然后你就可以选择所有或者部分项，然后切换到 Privileges（权限）选项卡上以设置你想要授予的角色和权限。

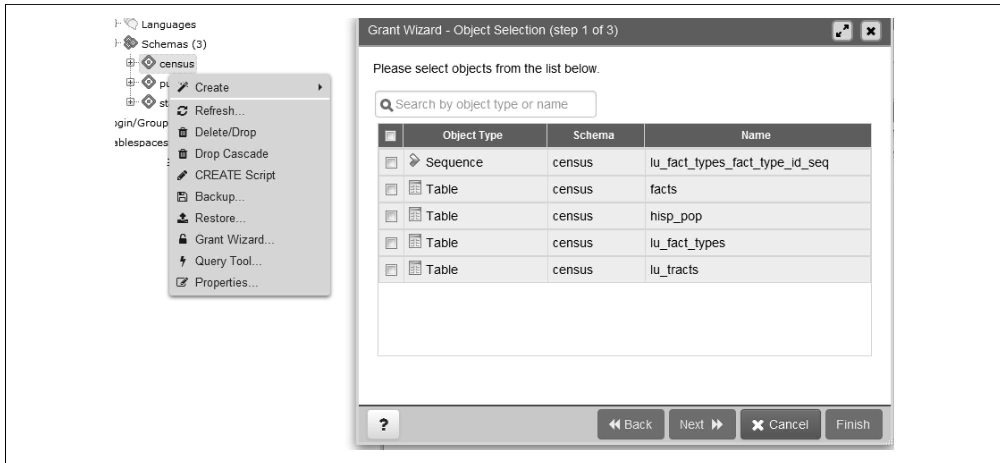


图 4-8：pgAdmin4 的授权向导

除了为已有对象授权以外，我们日常遇到更多的一个场景是为一个 schema 或者 database 中新建的对象设置默认权限。要执行此类授权，请右键单击 schema 或者 database 对象节点，然后选择 Properties（属性）菜单项，然后在弹出的界面上点击切换到 Default Privileges（默认权限）选项卡，如图 4-9 所示。

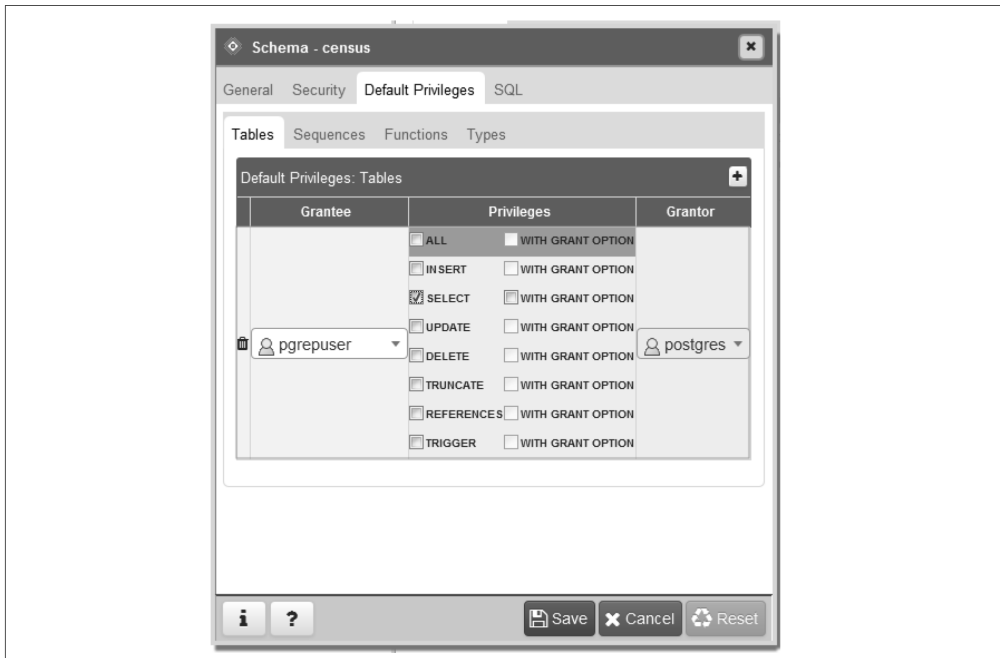


图 4-9：pgAdmin4 的默认授权管理

当为 schema 授予默认权限时，请记得一定要为相应的组角色授予访问此 schema 的权限。

4.2.5 数据导入和导出

同 psql 一样，pgAdmin 也可以导入和导出文本文件。

1. 导入文件

pgAdmin 的导入功能其实是对 psql 的 `\copy` 命令做了一层封装，并要求导入数据的目的表必须已建好。要实现数据导入，请在要导入数据的表上单击鼠标右键。图 4-10 显示了我们在 `lu_fact_types` 表上点击右键后出现的界面。

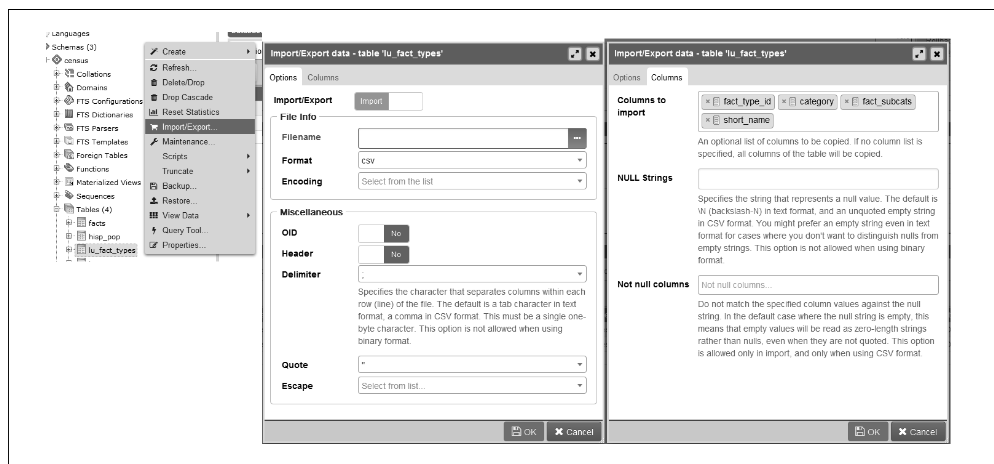


图 4-10：pgAdmin4 的 Import（导入）菜单

2. 使用pgAdmin将数据导出为结构化文件或者报表格式

除了导入数据，你还可以将查询结果导出。pgAdmin3 支持导出为 CSV、HTML 或者 XML 格式。pgAdmin4 的导出功能比 pgAdmin3 要弱很多。

要使用 pgAdmin 导出分隔符文本格式，请按以下步骤操作。

(1) 打开查询窗口（Query Tool）。
(2) 编写查询语句。
(3) 执行查询语句。
(4) 如果是 pgAdmin3，请点击菜单栏上的 File → Export；如果是 pgAdmin4，请点击下载图标（ ）并设定下载目标位置。
(5) 如果是 pgAdmin3，在跳出保存页面前会给出一个提示，pgAdmin4 则没有。按照图 4-11 填写设置内容。

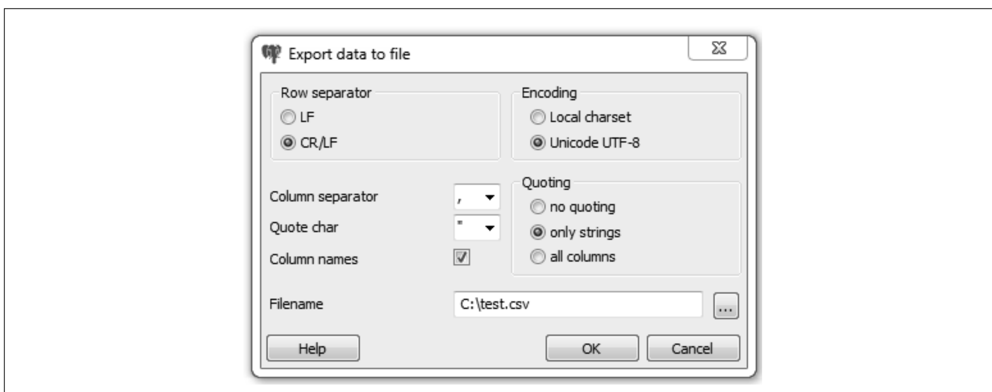


图 4-11：Export（导出）菜单

导出为 HTML 或者 XML 的步骤非常类似，唯一的差别在于需要点击菜单栏上的 File → Quick Report（快速报表）选项，参见图 4-12。

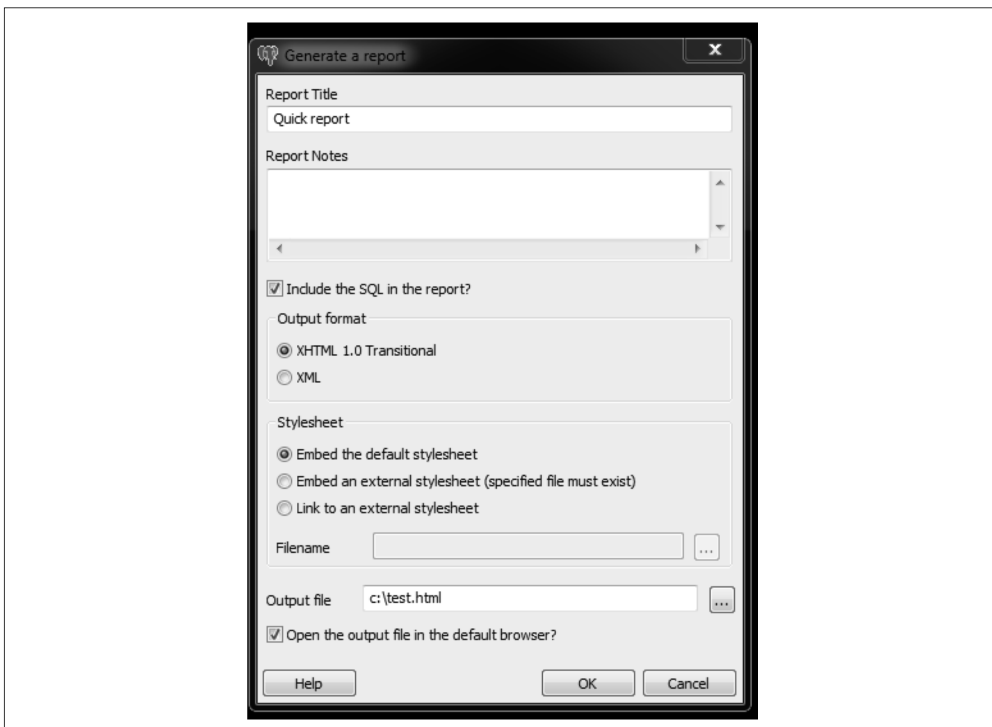


图 4-12：导出报表选项

4.2.6 备份与恢复

pgAdmin 为 `pg_dump` 和 `pg_restore` 提供了图形化的操作界面，相关具体功能已在 2.7 节中

介绍过。本节内容中，我们将重复使用一些前面已经使用过的例子，不过是使用 pgAdmin 来执行操作，而非使用命令行。

如果你的机器上安装了多个版本的 PostgreSQL 或 pgAdmin，建议你先确认 pgAdmin 指向了正确版本的 PostgreSQL 的 bin 目录（即 `pg_dump`、`pg_restore` 等命令行工具所在的目录），可以通过检查 pgAdmin 的 bin 目录设置来确认，如图 4-13 所示。

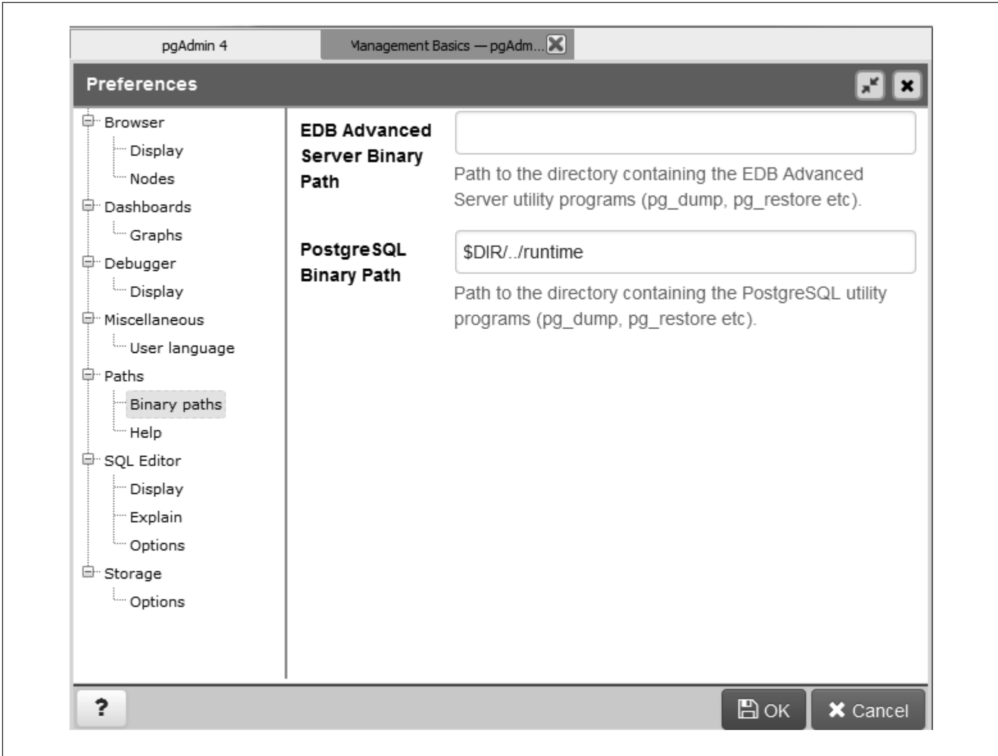


图 4-13：pgAdmin File → Preferences 菜单



如果你是在对一台远程服务器进行备份或者恢复操作，或者你操作的数据库数量特别庞大，那么建议你使用命令行工具而不要使用 pgAdmin，因为此种情况下操作过程本来就已经很耗时，使用 pgAdmin 操作会使得耗时和复杂度增加。另外也请牢记：对于 `pg_dump` 转储的自定义压缩格式、TAR 包格式、目录格式这三种二进制格式的备份文件，必须使用与 `pg_dump` 相同或者更新版本的 `pg_restore` 工具来进行恢复。

1. 完整备份一个database中的数据

在 2.7.1 节中，我们已经演示了如何完整备份一个 database。下面使用 pgAdmin 界面再演示一遍操作过程。请右键单击待备份的 database，并选择 Custom（自定义）格式，如图 4-14 所示。



图 4-14：备份 database

2. 备份系统级对象

pgAdmin 为 `pg_dumpall` 提供了一个图形化界面，用于对系统对象进行备份。要使用该界面，请先连接到希望备份的 PostgreSQL 服务器。然后从顶部菜单中选择 Tools（工具）→ Backup Globals（全局备份）。

pgAdmin 不支持指定备份哪些全局对象，但在 `pg_dumpall` 的命令行界面上是可以的。pgAdmin 默认会备份所有的系统表空间和角色。

如果你希望备份整个服务器端的所有数据，可以通过点击菜单栏上的 Tools → Backup Server（备份服务器）来实现。

3. 选择性地备份部分数据库对象

pgAdmin 为 `pg_dump` 的选择性备份功能提供了一个图形化接口。在希望备份的数据库对象上右键单击，然后在弹出的菜单中选择 Backup（如图 4-15 所示）。你可以选择备份整个 database、一个特定的 schema 或者任何其他数据库对象。

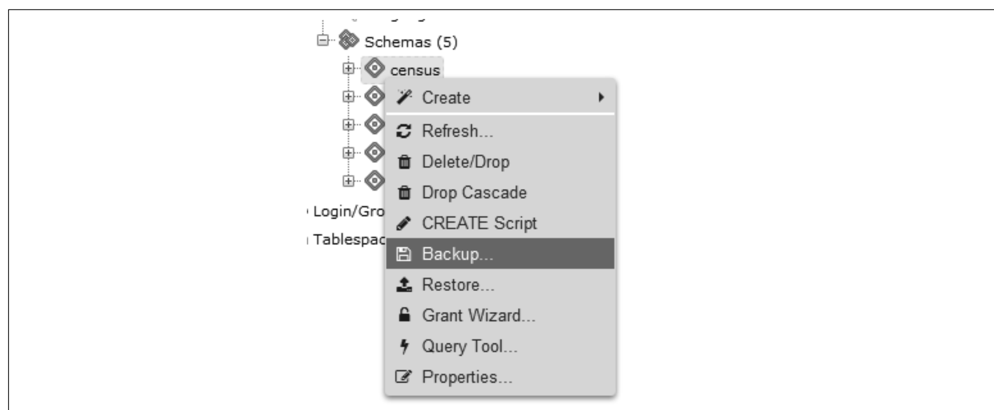


图 4-15：pgAdmin 的 schema 备份

在 pgAdmin3 中，如果希望仅备份当前图形界面上选中的数据库对象，那么你可以忽略备份界面上的其他选项卡（如图 4-14 所示），只用默认设置即可。当然，你也可以切换到 Objects（对象）选项卡选择备份更多对象，如图 4-16 所示。注意该特性在 pgAdmin4 中还不支持。

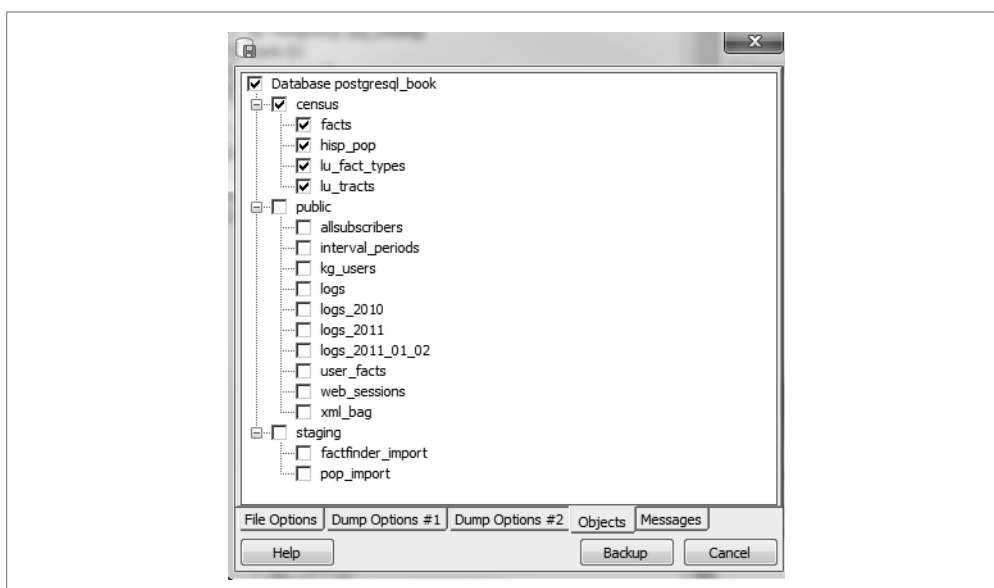


图 4-16：pgAdmin3 的选择性备份 Objects 选项卡



pgAdmin 在后台其实就是调用了 `pg_dump` 命令行工具来实施备份动作，如果你希望了解 pgAdmin 最终使用的命令是什么样子，那么可以在点击 Backup 按钮开始执行备份后切换到备份界面最右侧的 Messages（消息）选项卡，其中会记录系统自动生成的带参数的 `pg_dump` 命令行。

4.3 pgScript脚本机制

pgScript 是 pgAdmin3 内置的一种脚本机制，pgAdmin4 还不支持此特性。该机制非常适合于重复执行 SQL 任务的场景。相比 PostgreSQL 的函数机制，pgScript 对内存的使用更合理，因而执行效率更高。pgScript 机制之所以能达到这种效果，是因为 PostgreSQL 函数机制会将所有工作在最后一次性批量提交，此前未提交的工作成果都保存在内存中。相比之下，pgScript 在运行脚本时每执行一条 SQL 语句就提交一次，这样就使得 pgScript 特别适合执行会消耗大量内存而又不需要作为一个完整事务来提交的任务。一旦某个事务被提交，则该事务占用的内存会立即被释放，这部分内存即可用于下一个事务。在“Using pgScript for Geocoding”这篇博文中你可以看到我们所做的示例。

pgScript 脚本语言是一种弱类型语言（即定义变量时无须明确指定其类型），支持条件判断、循环、数据生成器、基本的打印函数以及记录型变量，其语法与微软 SQL Server 数据库所使用的 Transact-SQL 语法类似。前面加 `@` 的是变量，可以存放标量或数组，包括 SQL 命令的执行结果。`DECLARE`、`SET`、`IF-ELSE`、`WHILE` 等语法在 pgScript 中都支持。

可以在 SQL 查询执行窗口中执行 pgScript。在窗口中输入脚本后，点击 pgScript 图标来执行（）。

下面将演示一些 pgScript 脚本的例子。示例 4-1 演示了如何使用 pgScript 记录型变量和循环语法来构建一个交叉表，使用的基础表是 `lu_fact_types`，该表是我们在示例 7-22 中创建的。以下 pgScript 脚本中创建了一个名为 `census.hisp_pop` 的空表，该表有以下数字型列：`hispanic_or_latino`、`white_alone` 和 `black_or_african_american_alone`，以及其他一些列。

示例 4-1　在 pgScript 中使用记录型变量建表

```
DECLARE @I, @labels, @tdef;
SET @I = 0;

变量labels将用于存放记录
SET @labels =
    SELECT
        quote_ident(
            replace(
                replace(lower(COALESCE(fact_subcats[4], fact_subcats[3])), ' ',
'_'),':',''
            )
        ) As col_name,
    fact_type_id
    FROM census.lu_fact_types
    WHERE category = 'Population' AND fact_subcats[3] ILIKE 'Hispanic or Latino%'
    ORDER BY short_name;

SET @tdef = 'census.hisp_pop(tract_id varchar(11) PRIMARY KEY ';

使用LINES函数来循环遍历每一条记录
WHILE @I < LINES(@labels)
BEGIN
    SET @tdef = @tdef + ', ' + @labels[@I][0] + ' numeric(12,3) ';
    SET @I = @I + 1;
END
```

```
SET  @tdef = @tdef + ')';

打印表def
PRINT @tdef;

创建表
CREATE TABLE @tdef;
```

尽管 pgScript 中没有专门用于执行动态 SQL 的命令，但我们可以通过示例 4-1 中所示的方法来实现执行动态 SQL，即将 SQL 字符串分配给某个变量。我们在示例 4-2 中更加深入地挖掘了 pgScript 的功能，该示例中我们对上面刚刚创建好的 census.hisp_pop 表进行了填充操作。

示例 4-2　使用 pgScript 循环填充表

```
DECLARE @I, @labels, @tload, @tcols, @fact_types;
SET @I = 0;
SET @labels =
    SELECT
        quote_ident(
            replace(
                replace(
                    lower(COALESCE(fact_subcats[4], fact_subcats[3])), ' ',
'_'),':',''
            )
        ) As col_name,
    fact_type_id
    FROM census.lu_fact_types
    WHERE category = 'Population' AND fact_subcats[3] ILIKE 'Hispanic or Latino%'
    ORDER BY short_name;

SET @tload = 'tract_id';
SET @tcols = 'tract_id';
SET @fact_types = '-1';

WHILE @I < LINES(@labels)
BEGIN
    SET @tcols = @tcols + ', ' + @labels[@I][0] ;
    SET @tload = @tload +
        ', MAX(CASE WHEN fact_type_id= ' +
        CAST(@labels[@I][1] AS STRING) +
        ' THEN val ELSE NULL END)';
    SET @fact_types = @fact_types + ', ' + CAST(@labels[@I][1] As STRING);
    SET @I = @I + 1;
END

INSERT INTO census.hisp_pop(@tcols)
SELECT @tload FROM census.facts
WHERE fact_type_id IN(@fact_types) AND yr=2010
GROUP BY tract_id;
```

从上面的示例中可以学到的一点是：可以动态地往一个变量中一点点加入 SQL 语句的零碎部分，最终组成一个完整的 SQL 语句。

4.4　以图形化方式解释执行计划

pgAdmin 最为人称道的优秀功能之一就是它能够以图形化方式展示语句执行计划。打开 SQL 语句执行窗口，编写一个 SQL 语句，然后点击“解释查询”图标（ ）就可以看到此语句的执行计划图示。

例如，执行以下查询：

```
SELECT left(tract_id, 5) As county_code, SUM(hispanic_or_latino) As tot,
    SUM(white_alone) As tot_white,
    SUM(COALESCE(hispanic_or_latino,0) - COALESCE(white_alone,0)) AS non_white
FROM census.hisp_pop
GROUP BY county_code
ORDER  BY county_code;
```

我们将看到如图 4-17 所示的图形化解释。读懂这种图形化解释有一个小窍门，那就是尽可能让粗箭头变细！箭头越粗，说明该步骤的执行时间越长。

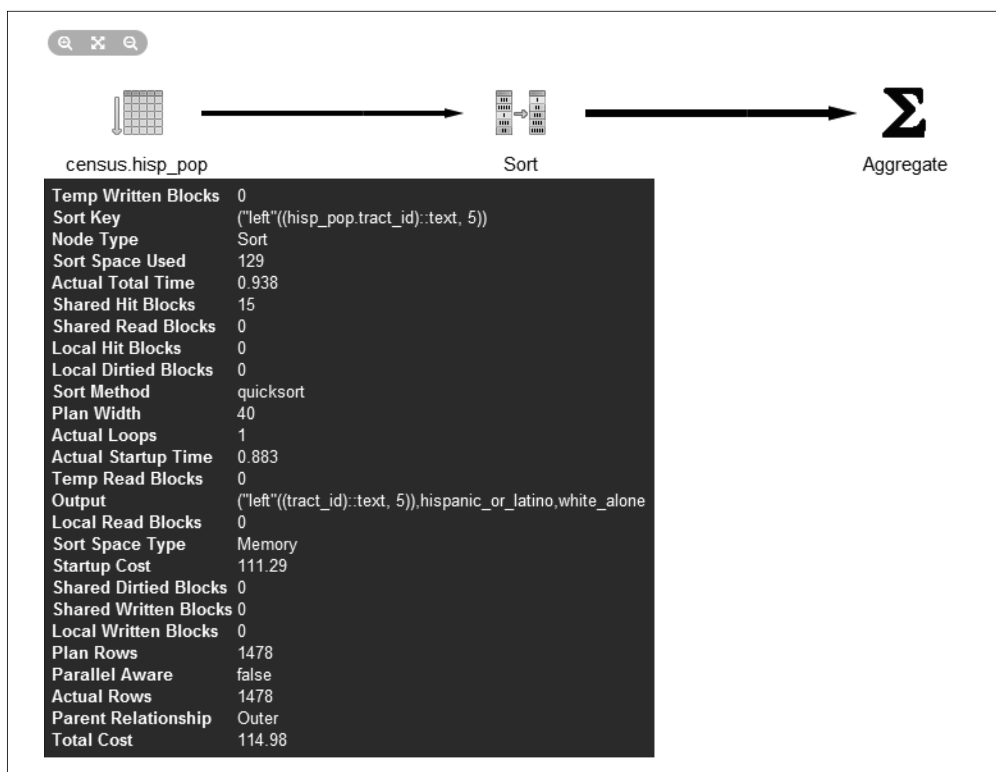


图 4-17：图形化解释示例

如果 SQL 语句执行器菜单栏上的 Query（查询）→ Explain（解释）→ Buffers（缓冲区）已启用，那么图形化解释就会被禁用。因此切记使用图形化解释时不要启用该选项。除了图形化解释之外，Data Output（数据输出）选项卡上还会显示文本解释计划，本例中的输

出如下所示。

```
GroupAggregate  (cost=111.29..151.93 rows=1478 width=20)
  Output: ("left"((tract_id)::text, 5)), sum(hispanic_or_latino),
sum(white_alone), ...
  -> Sort   (cost=111.29..114.98 rows=1478 width=20)
    Output: tract_id, hispanic_or_latino, white_alone,
    ("left"((tract_id)::text, 5))
       Sort Key: ("left"((tract_id)::text, 5))
       ->  Seq Scan on census.hisp_pop  (cost=0.00..33.48 rows=1478 width=20)
                 Output: tract_id, hispanic_or_latino
                  , white_alone, "left"((tract_id)::text, 5)
```

4.5 使用pgAgent执行定时任务

pgAgent 是 PostgreSQL 中执行定时任务的得力工具。同时它也可用于执行操作系统批处理脚本，因此在 Linux/Unix 系统中它可取代 `crontab`；在 Windows 中，它可取代定时任务规划器。事实上，pgAgent 的定时任务功能远比这里描述的更强大：任何一台机器，不管操作系统是什么，只要它上面能安装 pgAgent，我们就可以在其上执行定时任务。具体步骤是先在这台机器上装好 pgAgent，然后设置该 pgAgent 连接到一个 PostgreSQL 数据库上，但需要该数据库上预先安装好 pgAgent 所需的功能表和函数。执行定时任务的机器上不需要安装 PostgreSQL 服务端软件，但客户端数据库是必须的，因为要保证 pgAgent 能够连接到外部的 PostgreSQL 服务器。pgAgent 是构建于 PostgreSQL 架构基础上的，因此你可以通过控制服务端的表数据来控制 pgAgent 的行为。例如，如果需要将一个复杂的定时任务复制多次，那么你只需直接登录到 PostgreSQL 服务器并往 pgAgent 的表中插入几条记录即可，完全不需要在 pgAmin 界面上对 pgAgent 执行操作。

本节将教你如何使用 pgAgent。在“Setting Up pgAgent and Doing Scheduled Backups”这篇博文中你可以看到一个真正实用的例子以及设置细节。

4.5.1 安装pgAgent

你可以从 http://www.pgadmin.org/download/pgagent.php 下载 pgAgent 的安装包。在 Windows 上，你也可以通过 EnterpriseDB 公司提供的 Stackbuilder 软件和 BigSQL 公司的发行包来安装 pgAgent。安装包中的 SQL 脚本会在 `postgres` 库中自动创建一个名为 pgAgent 的新 schema。然后当你通过 pgAdmin 连到数据库服务器时，可以在目录树上看到一个名为 Jobs（作业）的新节点，如图 4-18 所示。

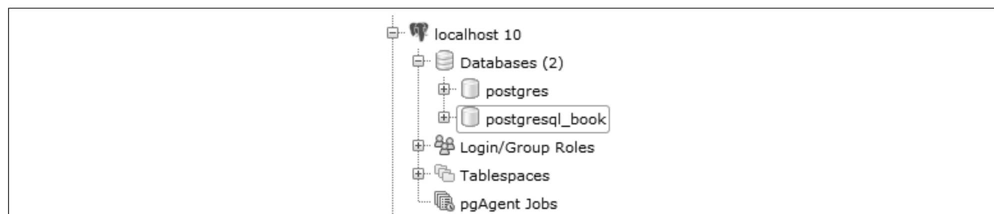


图 4-18：安装了 pgAgent 后的 pgAdmin4 界面

如果你希望在其他机器上安装 pgAgent 来执行定时任务，仅需在目标机器上安装 pgAgent 客户端即可，而无须再次执行 pgAgent 自带的 SQL 脚本，因为这个脚本只需在 PostgreSQL 服务端执行一次即可。请特别注意 pgAgent 服务所使用的操作系统账户的权限设置，一定要确保每个 pgAgent 客户端实例都有执行任务所需的权限。



即使批处理任务能以命令行方式执行成功，也并不代表通过 pgAgent 来执行此任务就一定会成功。这一般是因为权限原因导致，pgAgent 总是以 pgAgent 服务所使用的操作系统账户的身份执行规划的任务，如果此账户没有足够的权限来执行此批处理任务或者没有访问某些必需路径的权限，那么任务就会失败。

4.5.2 规划定时任务

每个定时任务包含两个组成要素：任务步骤以及执行计划。当创建一个新的任务时，先新增一个或者多个任务步骤。图 4-19 是新增 / 编辑任务步骤的界面。

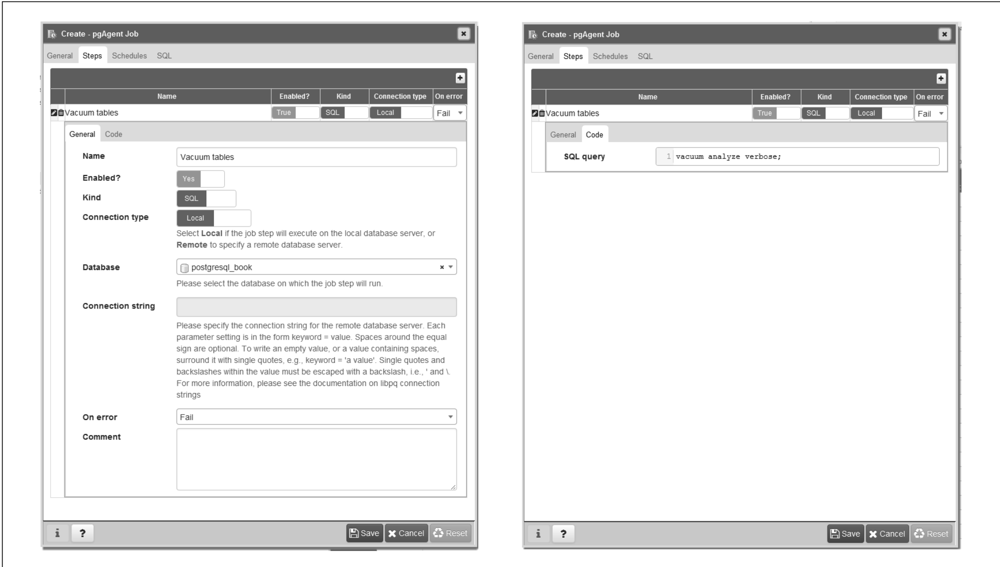


图 4-19：pgAdmin 的分步编辑屏幕

在每一个任务步骤中，你可以设定一条 SQL 语句或者指定一个 shell 脚本作为任务内容，甚至可以复制一整段 shell 脚本作为任务内容。

如果你选择了使用 SQL 语句，那么连接类型选项会变为可用并且默认值会被设为本地，这种情况下该任务步骤中的 SQL 会在 pgAgent 服务端所在的 PostgreSQL 服务器上执行，并使用 pgAgent 运行时使用的用户名和密码来进行身份验证。另外还需要指定 pgAgent 在执行此任务时要连接的目标 database。界面上有一个下拉列表供你选择具体的 database。如果你选择了连接到远端数据库服务器，那么供输入连接字符串的文本框会变为可用状态。请

在此框中输入完整的连接字符串，包括用于身份验证的信息以及要连接的目标 database。如果你连到一个早期版本的远端 PostgreSQL 数据库，请确保要执行的 SQL 语句的语法在该老版本的 PostgreSQL 上是支持的。

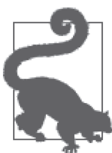
如果你选择了执行批处理任务，那么其语法必须符合执行此任务的操作系统的要求。例如，如果 pgAgent 运行于 Windows 环境下，那么批处理任务脚本必须是合法的 DOS 命令行脚本；如果 pgAgent 运行于 Linux 环境，那么批处理任务脚本必须是合法的 shell 脚本。

多个任务步骤之间是以其名称的字母顺序来排序并执行的。你可以指定每个步骤执行完毕后的处理方式：如果该步骤执行成功则如何处理；如果该步骤执行失败又如何处理。你可以选择禁用某些步骤而不删除它们，因为你以后可能会重新用上这些步骤。

任务步骤设定好之后，你就可以设定执行计划来执行这些步骤了。通过执行计划页面你可以设定极为复杂的执行策略，你甚至可以设置多个执行计划。

如果你在多台机器上安装了 pgAgent，而这些 pgAgent 都连到同一个 pgAgent 服务端数据库，那么默认情况下所有这些 pgAgent 会执行数据库中记录的所有计划任务。

如果你希望某个计划任务只在某台特定的机器上执行，那么可以在创建计划任务时将页面上的 `host agent`（主机代理）字段设置为希望执行此计划任务的目标主机名。这样其他机器上的 pgAgent 会发现此计划任务的目标主机名与自己所在主机名不符，从而忽略此任务。



pgAgent 包含两部分数据：定义任务的数据以及任务执行日志。这些任务日志会记录在 pgAgent 这个 schema 中，而 pgAgent schema 一般隶属于 `postgres` 数据库。pgAgent 进程会查询待执行的任务信息以决定接下来执行什么任务，然后在执行过程中把相关的任务日志信息写入数据库中。一般来说，用于承载这两类数据的 PostgreSQL 服务器和 pgAgent 是运行于同一台服务器上的，但并不是必须如此，二者可以分离部署。此外，一台 PostgreSQL 服务器可以服务于很多部署在不同主机上的 pgAgent。

一个完整的定时任务看起来如图 4-20 所示。

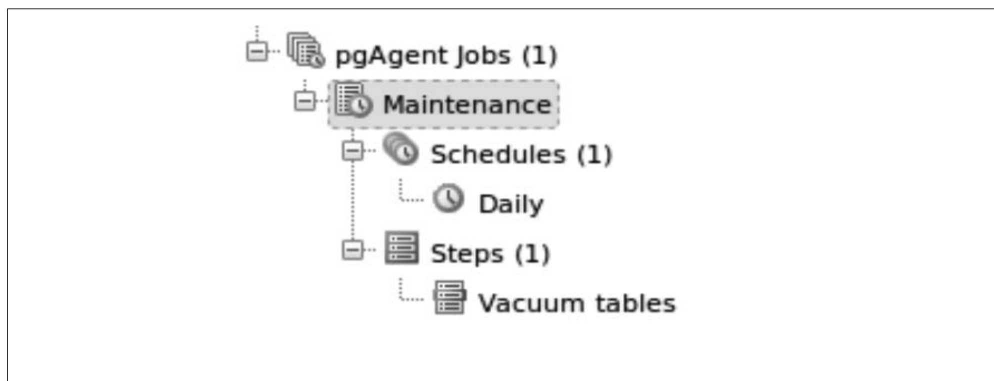


图 4-20：pgAdmin 界面上的 pgAgent 定时任务

4.5.3　一些有用的pgAgent相关查询语句

如果你 SQL 技术高超，那么完全可以通过直接修改 pgAgent 的元数据表来实现对定时任务的复制、删除和修改。只是在修改时要特别小心谨慎，不要搞错！例如，要想查看控制着 pgAgent 和定时任务具体行为的后台表内容，只需连到 `postgres` 数据库并执行示例 4-3 中的查询。

示例 4-3　查询 pgAgent 相关表的描述信息

```
SELECT c.relname As table_name, d.description
FROM
    pg_class As c INNER JOIN
    pg_namespace n ON n.oid = c.relnamespace INNER JOIN
    pg_description As d ON d.objoid = c.oid AND d.objsubid = 0
WHERE n.nspname = 'pgagent'
ORDER BY c.relname;

table_name     |       description
---------------+------------------------
pga_job        | Job main entry
pga_jobagent   | Active job agents
pga_jobclass   | Job classification
pga_joblog     | Job run logs.
pga_jobstep    | Job step to be executed
pga_jobsteplog | Job step run logs.
pga_schedule   | Job schedule exceptions
```

尽管在 pgAdmin 中制定 pgAgent 定时任务和观察其执行日志的界面已经非常直观易懂，但如果你设置了很多定时任务或者你希望查看自己的定时任务执行结果总览，那么就需要生成自己的定时任务执行报告。示例 4-4 演示了我们在此情况下经常会使用的一个查询语句。

示例 4-4　列出从今天开始的定时任务执行结果

```
SELECT j.jobname, s.jstname, l.jslstart,l.jslduration, l.jsloutput
FROM
    pgagent.pga_jobsteplog As l INNER JOIN
    pgagent.pga_jobstep As s ON s.jstid = l.jsljstid INNER JOIN
    pgagent.pga_job As j ON j.jobid = s.jstjobid
WHERE jslstart > CURRENT_DATE
ORDER BY j.jobname, s.jstname, l.jslstart DESC;
```

有时候定时任务即使失败了也会报成功，因为 pgAgent 并不总能准确判断 shell 脚本的执行结果到底是成功还是失败。日志中的 `jsloutput` 字段提供 shell 输出，该输出通常会详细说明哪里出现了错误。



在 Windows 平台上，有若干个版本的 pgAgent 经常会把事实上已执行成功的 shell 脚本的执行结果判定为执行失败。如果你遇到了这种情况，那么应该把任务步骤的结果处理方式设定为“错误时忽略”。这是一个已知的 bug，我们希望在将来的版本中能够尽快修复。

第 5 章

数据类型

与其他所有数据库一样，PostgreSQL 也支持数字型、字符串型、日期型、时间型以及布尔型等业界常用的数据类型。但 PostgreSQL 的先进之处在于它还支持数组、带时区的日期时间、时间间隔、区间、JSON、XML 以及其他很多数据类型，此外还支持用户自定义数据类型。本章不会一一介绍 PostgreSQL 支持的所有数据类型，如果你需要了解的话，请自行参考官方手册。我们将着重介绍 PostgreSQL 独有的若干数据类型，以及在通用数据类型方面 PostgreSQL 与其他数据库有哪些细微差异。

如果离开了相应的函数和运算符，数据类型将完全无用武之地。PostgreSQL 为各种数据类型提供了功能强大、种类丰富的原生函数和运算符支持。本章将介绍一些使用比较广泛的类型。



我们所说的**函数**是指 `f(x)` 这种形式的函数；我们所说的“运算符”是指一些符号性的运算符号，比如 `+`、`-`、`*`、`/`，具体可分为需要单个参数的一元运算符和需要两个参数的二元运算符。当使用运算符时，请记住它在面对不同的数据类型时所代表的含义是不同的。比如加号对数字来说就是相加，对区间类型来说就是区间的并集。

5.1 数值类型

PostgreSQL 支持常用的整数、小数、浮点数等数字类型。本节想重点介绍的是 `serial` 类型以及一个灵活实用的整数序列生成函数。

5.1.1 serial类型

`serial` 类型和它的兄弟类型 `bigserial` 是两种可以自动生成递增整数值的数据类型，一般

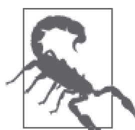
如果表本身的字段不适合作为主键字段，会增加一个专门的字段并指定为 serial 类型以作为主键。在不同的数据库产品中这种数据类型有着不同的称呼，一般最常见的叫法是 autonumber。创建表时如果指定了一个字段类型为 serial，那么 PostgreSQL 会首先将其作为整型处理，同时自动在该表所在 schema 中创建一个名为 table_name_column_name_seq 的序列。然后设定该序列为该整型字段的取值来源。如果修改了表定义并删除此 serial 字段，那么系统同时也会自动删除附属的序列。

在 PostgreSQL 中，序列自身就是一种数据库资产。可以通过 pgAdmin 图形界面或者 ALTER SEQUENCE 语句来管理该类对象。可以设置其当前值和边界值（也就是最大值和最小值），还可以设置每次递增的步长。虽然一般来说序列值都是递增的，但你也可以将其设为递减，只要将步长值 increment 设为负数即可。作为一种独立的数据库资产，你可以通过 CREATE SEQUENCE 命令来创建序列，还可以在多张表间共用一个序列。如果需要生成一个跨越多表的唯一键值，那么这种多表共享序列的用法特别方便。

如需多表共享同一个现存的序列号生成器，先为每张表新增一个类型为 integer 或者 bigint 的字段，然后指定其默认值为 nextval(sequence_name) 即可，语法如示例 5-1 所示。

示例 5-1　在新表上使用现存的序列号生成器

```
CREATE SEQUENCE s START 1;
CREATE TABLE stuff(id bigint DEFAULT nextval('s') PRIMARY KEY, name text);
```



如果你重命名了一张含 serial 字段的表，这张表的 serial 字段关联的序列号生成器是不会跟着改名的，但关联运作机制是不受影响的。为避免混淆，可以手动修改序列号生成器的名称以保持与表名一致。

5.1.2　生成数组序列的函数

PostgreSQL 有一个名为 generate_series 的灵活又实用的数组生成函数，目前为止我们还没发现有哪种数据库支持类似功能。该函数支持两种使用形式。它可以用来生成一个按一定步长递增的整数序列，也可以用来生成一个以一定时间间隔作为步长来递增的日期或者时间戳序列。generate_series 函数的方便之处在于，你可以使用它有效地模仿 SQL 中的 for 循环。示例 5-2 演示了如何生成一个整数序列。示例 5-13 演示了如何生成时间序列。

示例 5-2 使用可选的步长参数来生成整数序列。

示例 5-2　使用 generate_series() 函数生成步长为 13 的整数序列

```
SELECT x FROM generate_series(1,51,13) As x;

x
----
1
14
27
40
```

如果不输入步长，则默认步长为 1。如示例 5-2 所示，你可以传入一个可选的步长参数来指定对于每个后续元素要跳过多大的步长。另外请注意：结束值将永远不会超出我们指定

的区间，因此，尽管我们的区间结束于 51，但最后一个数字是 40，因为 40 再加上 13 就会超出上限。

5.2 文本类型

PostgreSQL 有三种最基础的文本数据类型：character（也称为 char）、character varying（也称为 varchar）和 text。

仅当需要存储邮政编码、电话号码以及美国的社会保险号等定长字符串时，才应使用 char 类型。如果存储的字符长度达不到 char 类型的定义长度，那么 PostgreSQL 会自动在后面用空格填充，直到填满定义的长度为止。相比 varchar 或者 text，char 的右补齐空格机制会导致存储空间的浪费，得到的好处是可以确保字符串是定长的。可以确认的是，char 比起 varchar 和 text 没有任何性能优势，却一定会占用更多的空间。如果要存储变长字符串，请使用 varchar 类型。当为表定义 varchar 类型的字段时，应为其设置最大长度。text 类型是最通用的字符存储类型，不需要设最大长度。

varchar 类型的最大长度其实是可选的。如果不设定的话，varchar 与 text 几乎没有任何区别，但当通过一些特定的驱动连到 PostgreSQL 时，二者还是能表现出一些微小的差异。比如，ODBC 驱动可以对 varchar 字段进行排序，却不支持 text 类型的排序。varchar 和 text 类型字段的存储空间上限均为约 1GB，这是一个很大的值了。事实上，系统在后台会用 TOAST 机制处理超过一个物理存储页大小的内容。

有人认为应该彻底废弃 varchar 并完全转用 text，关于这一点，业界一直有着不同的声音。此处不会浪费篇幅争论此问题，请参考“In Defense of Varchar(X)”这篇博文以了解这场争论的详情。

有时候，为了保持跨平台应用的兼容性，需要使字符串类型的操作变得不区分大小写。要实现此目标，需要重写那些区分大小写的比较运算符。相比 text，对 varchar 进行运算符重写要更容易一些。我们在“Using MS Access with PostgreSQL”这篇博文中，演示了如何使 varchar 类型变得不区分大小写而且还能够用上索引。

5.2.1 字符串函数

常见的字符串操作包括：填充（lpad、rpad）、修整空白（rtrim、ltrim、trim、btrim）、提取子字符串（substring）以及连接（||）。示例 5-3 演示了填充操作，示例 5-4 演示了修整空白操作。

示例 5-3　使用 lpad 和 rpad 进行填充操作

```
SELECT
    lpad('ab', 4, '0') As ab_lpad,
    rpad('ab', 4, '0') As ab_rpad,
    lpad('abcde', 4, '0') As ab_lpad_trunc; ❶

ab_lpad | ab_rpad | ab_lpad_trunc
--------+---------+---------------
00ab    | ab00    | abcd
```

❶ 如果字符串超过指定长度，`lpad` 不但不会填充，反而会对其进行截断。

默认情况下，`trim` 函数用于移除空格，但你也可以传入一个可选参数，指示要剪裁的其他字符。

示例 5-4　剪裁空格和字符

```
SELECT
    a As a_before, trim(a) As a_trim, rtrim(a) As a_rt,
    i As i_before, ltrim(i, '0') As i_lt_0,
    rtrim(i, '0') As i_rt_0, trim(i, '0') As i_t_0
FROM (
        SELECT repeat(' ', 4) || i || repeat(' ', 4) As a, '0' || i As i
        FROM generate_series(0, 200, 50) As i
) As x;

a_before     | a_trim |  a_rt   | i_before | i_lt_0 | i_rt_0 | i_t_0
-------------+--------+---------+----------+--------+--------+-------
0            | 0      |  0      | 00       |        |        |
50           | 50     |  50     | 050      | 50     | 05     | 5
100          | 100    |  100    | 0100     | 100    | 01     | 1
150          | 150    |  150    | 0150     | 150    | 015    | 15
200          | 200    |  200    | 0200     | 200    | 02     | 2
```

有一个很有用的字符串操作函数 `string_agg`，我们已在示例 3-14 中展示了其用法，你还将在示例 5-26 中再次见到它。

5.2.2　将字符串拆分为数组、表或者子字符串

PostgreSQL 中有一些函数可以对字符串进行拆分操作。

`split_part` 函数可以将指定位置的元素从用固定分隔符分隔的字符串中取出来，如示例 5-5 所示。这里我们取出用圆点分隔的字符串中的第二个元素。

示例 5-5　取出分隔符字符串中的第 *n* 个元素

```
SELECT split_part('abc.123.z45','.', 2) As x;

x
---
123
```

`string_to_array` 函数可以将基于固定分隔符的字符串拆分为一个数组。通过结合使用 `string_to_array` 和 `unnest` 函数，可以将一个字符串展开为若干记录行，如示例 5-6 所示。

示例 5-6　将基于固定分隔符格式的字符串展开为记录行

```
SELECT unnest(string_to_array('abc.123.z45', '.')) As x;

x
---
abc
123
z45
```

5.2.3 正则表达式和模式匹配

PostgreSQL 对正则表达式的支持是极其强大的。你可以设定查询返回结果的格式为表或者数组，并且对其进行极其复杂的替换和更新操作。包括逆向引用（back reference）在内的一些高级搜索方法都是支持的。本节将提供一个小型的示例以说明这些内容。如需了解更多相关信息，请参考 PostgreSQL 官方手册中的“模式匹配”和“字符串函数”这两节的内容。

示例 5-7 展示了如何对以数字形式存储的电话号码进行格式化操作。

示例 5-7　使用逆向引用技术对电话号码进行重新格式化

```
SELECT regexp_replace(
'6197306254',
'([0-9]{3})([0-9]{3})([0-9]{4})',
E'\(\\1\) \\2-\\3'
) As x;

x
--------------
(619) 730-6254
```

`\\1` 和 `\\2` 是模式匹配表达式中的元素。我们使用反斜杠（`\`）来转义圆括号，表示此处是真的要作为一个圆括号来用。`E'` 是 PostgreSQL 的特有语法，表示后续跟着的字符串是一个表达式，其中类似 \ 的特殊字符应该按照字面含义来处理。

假设一个字符串中内嵌了一些电话号码，示例 5-8 演示了如何仅通过一个 SQL 语句就将这些号码提取出来并作为记录行输出。

示例 5-8 将文本中的电话号码作为单独的行返回

```
SELECT unnest(regexp_matches(
'Cell (619) 852-5083. Work (619)123-4567 , Casa 619-730-6254. Bésame mucho.',
E'[(]{0,1}[0-9]{3}[)-.]{0,1}[\\s]{0,1}[0-9]{3}[-.]{0,1}[0-9]{4}', 'g')
) As x;

x
--------------
(619) 852-5083
(619)123-4567
619-730-6254
(3 rows)
```

示例 5-8 中用到的匹配规则如下所示。

- `[(]{0,1}`：开始是 0 个或者 1 个 `(`。
- `[0-9]{3}`：跟着 3 位数字。
- `[)-.]{0,1}`：跟着 0 个或者 1 个 `)`、`-` 或者 `.`。
- `[0-9]{4}`：跟着 4 位数字。
- `regexp_matches` 函数会返回根据一个正则表达式筛选匹配得到的字符串数组。该函数的最后一个入参名为 `flags`，我们为其输入的值是 `g`，`g` 代表 global，即需要进行完整搜索并返回所有匹配上的字符串，每个字符串作为数组中的一个元素。如果不填该 `flags` 参数，那么返回的结果只会包含第一个命中的字符串。请注意，`flags` 参数中可以包含多

个标记。例如，如果你的正则表达式中含有一些字符，你又希望在做正则匹配时不区分这些字符的大小写（即不管大小写都命中），同时还希望返回所有匹配上的字符串而不仅仅返回第一个，那么可以用 gi 这个复合标记。除此以外还支持其他一些标记，完整的列表可以从官方手册的“POSIX 标准中的嵌入式选项”部分查到。

- unnest 函数将一个数组分解为一个行集。



同一个正则表达式可以有多种写法。比如，\\d 代表 [0-9]。但我们建议不要为了省那几个字符而把表达式搞得太晦涩难懂，应该采用更加容易理解的写法。

如果只需要命中第一条，那么可以使用 substring 函数，如示例 5-9 所示。

示例 5-9　从一段文本中查找第一个电话号码

```
SELECT substring(
'Cell (619) 852-5083. Work (619)123-4567 , Casa 619-730-6254. Besame mucho.'
from E'[(]{0,1}[0-9]{3}[)-.]{0,1}[\\s]{0,1}[0-9]{3}[-.]{0,1}[0-9]{4}')
 As x;

      x
----------------
 (619) 852-5083
(1 row)
```

除了正则表达式专用的那些函数外，你还可以将正则表达式与 SIMILAR TO（~）运算符一起使用。以下查询可以查出所有内嵌了电话号码的 description 字段：

```
SELECT description
FROM mytable
WHERE description ~
E'[(]{0,1}[0-9]{3}[)-.]{0,1}[\\s]{0,1}[0-9]{3}[-.]{0,1}[0-9]{4}';
```

5.3　时间类型

PostgreSQL 对时间类型的支持在业界是无人能及的。除了常见的日期和时间类型，PostgreSQL 还支持时区，并能够按照不同的时区对夏令时进行自动转换。此外 PostgreSQL 还支持一些特殊的数据类型，如 interval，该类型可以用于对日期时间进行数学运算。PostgreSQL 还有正无穷大和负无穷大的概念，这样我们就不用为了表达这两个概念而弄出一些奇奇怪怪的潜规则，搞这些潜规则迟早会导致问题。区间类型可以表达时间区间的概念，并且提供了大量与区间运算相关的运算符、函数和索引。5.5 节中会详细介绍该类型。

在最新的版本中，PostgreSQL 支持 9 种与时间相关的数据类型。理解这些类型之间的区别很重要，否则你就无法为不同业务场景选择适用的数据类型。除了 range 类型外，其他所有类型都遵循 ANSI SQL 标准。业界的其他数据库最多支持这些类型中的一部分而非所有。Oracle 支持的类型最多，SQL Server 其次，MySQL 又次之。

PostgreSQL 的每种时间类型都有其独特之处。对于其中支持时区的类型，如果数据库服务

器所在的时区发生了改变，则该类型中存储的数据会自动针对新的时区进行调整以保证时间一致。接下来对它们进行更详细的介绍。

`date`

该类型仅存储月、日、年，没有时区、小时、分和秒的信息。

`time`（又称 `time without time zone`）

该类型仅存储小时、分、秒信息，不带日期和时区信息。

`timestamp`（又称 `timestamp without time zone`）

该类型存储了日期（年、月、日）和时间（时、分、秒）数据，但不带时区信息。因此，即使你修改了数据库服务器所在的时区信息，该类字段查询出来显示的值也是固定不变的。

`timestamptz`（又称 `timestamp with time zone`）

该类型同时存储了日期、时间以及时区信息。在系统内部，该类型的字段值是以 UTC 世界标准时间格式存储的，但当查询显示时，会按照服务器的时区设置进行换算后再显示（时区也可以在库级 / 用户级 / 会话级分别进行设置）。如果你输入的时间戳不带时区数据，那么存入 `timestamptz` 类型字段中时，PostgreSQL 会自动使用当前数据库服务器的时区信息来补充。如果修改了数据库服务器的时区设置，你可以看到查询出来的时间数据发生了变化。

`timetz`（又称 `time with time zone`）

与 `timestamptz` 类型类似，但该类型的使用频率较低，因为它虽然携带了时区信息却没有日期信息。该类型永远假设当前时间是夏令时。有的编程语言不支持这种仅有时间而无日期的数据类型，因此可能会将其自动转换为带时区的时间戳类型，转换时日期就取计算机系统时间的初始值（例如，Unix 时间纪元起始于 1970 年，因此转换后的日期就是 1970 年 1 月 1 日，时区和时间不变，夏令时）。

`interval`

该类型描述了一个时间段的长度，单位可以是小时、天、月、分钟或者其他粒度。该类型适用于对日期和时间进行数学运算的场景。例如，假设从现在开始 666 天之后世界就会灭亡，那么你在现在的时刻上加上长度为 666 天的一个 `interval` 类型值，就可以知道世界灭亡的准确时刻。

`tsrange`

该类型可用于定义 `timestamp with no timezone` 的开区间和闭区间。该类型包含两个时间戳以及开区间和闭区间限定符。例如，`'[2012-01-01 14:00 2012-01-01 15:00)'::tsrange` 定义了从 14:00 开始到 15:00 之前结束的一个时间段。请参考 PostgreSQL 官方手册中“区间类型”一节以了解更多信息。

`tstzrange`

该类型可用于定义 `timestamp with timezone` 的开区间和闭区间。

`daterange`

该类型可用于定义日期的开区间和闭区间。

5.3.1 时区详解

PostgreSQL 中有众多支持时区的数据类型，关于它们有一个常见的误解，就是认为 PostgreSQL 会在日期和时间类型的基础上额外增加一个标记来标识时区。这种理解是错误的。如果你存储了这么一个带时区的信息：`2012-2-14 18:08:00-8`（-8 代表比 UTC 时间迟 8 小时的时区），PostgreSQL 内部其实是这么工作的：

(1) 通过计算得到 2012-02-14 18:08:00-8 代表的 UTC 标准时间，就是 2012-02-15 04:08:00-0；
(2) 把上述计算得到的 UTC 标准时间存储下来。

当你回调该数据以用于显示时，PostgreSQL 内部是这样运作的：

(1) 以字段内容中指定的时区作为后续计算的基础时区，如未指定，则用数据库服务器上配置的默认时区；
(2) 计算该时区相对于 UTC 标准时间的时差（对于 `America/New_York` 时区来说，与 UTC 的时差是 −5 小时）；
(3) 根据 UTC 标准时间和时区的时差计算出当地时间（2012-02-15 16:08:00 加上时差 −5 小时后得到 2012-02-15 21:08:00）；
(4) 显示计算结果（`2012-02-15 21:08:00`）。

可以看到，PostgreSQL 并没有存储时区信息，而只是使用时区信息来把日期和时间转换为 UTC 标准时间再存储下来。此后就不需要时区信息了。当 PostgreSQL 需要显示该日期时间信息时，它会按顺序查找当前会话级、用户级、数据库级、服务器级的时区设置，然后使用找到的第一个时区来将 UTC 标准时间转换为对应时区的时间值后再显示。如果你使用了带时区信息的数据类型，请务必了解将服务器从一个时区搬迁到另一个时区的后果。假设你的数据库服务器起初在纽约，然后你将其数据拿到洛杉矶做了恢复，那么所有带时区信息的日期和时间数据看起来都会不一样了。这乍看起来有点怪，但其实是正常的，你务必要预见到这种情况的发生。

下面将演示一个时区处理不当导致问题的例子。假设麦当劳公司的服务器都部署在东海岸，服务器中记录了各门店的开门营业时间，并且是用 `timetz` 格式存储的。然后旧金山开了一家新的麦当劳分店，分店经理给麦当劳总部打电话，要求把新店的信息纳入总部的管理数据库，并标记其开门营业时间为早上 7 点。于是位于东海岸的数据库服务器中会记录下该分店的营业时间为早上 7 点，但这个时间转换到旧金山当地时区却是凌晨 4 点。于是旧金山很多早起的人就会很奇怪，为什么明明说是凌晨 4 点开门却到了时间还没营业。买不到早点是小事，但你可以想象这三小时的时差能导致多么大的混乱，这甚至可能导致人命关天的问题。

看了上面的例子，你可能会问：既然这么危险，为什么还要使用带时区的时间类型？原因有以下几个。首先，这些类型能够自动执行时区转换，从而避免了繁琐的手工劳动。例如，某航空公司的某个航班早上 8 点从波士顿出发，11 点到达洛杉矶，但是该公司的数据库服务器位于欧洲，如果这些时间入库时都要手工计算时差后再录入，那就效率太低了。使用了支持时区的数据类型后，只需录入带时区信息的波士顿和洛杉矶本地时间即可。另一个使用带时区的数据类型的理由是其能够自动处理夏令时。世界各国对于夏令时的规定五花八门，如果某个数据库可能会被全球各地的应用访问，那么就需要及时按照最新的全

球夏令时规定来更新库中的时间信息。手动跟踪全球夏令时的变化是一件无比繁琐的工作，这需要一个全职的程序员来专门收集各国的夏令时安排，并在前述数据库中刷新这些国家（地区）的相关时间数据。

这里有一个非常有趣的例子：一位出差中的销售员需要坐飞机回家，起点是旧金山，终点是奥克兰附近。当她登上飞机时，当地时钟显示的时间是 2012 年 3 月 11 日凌晨 1 点 50 分。当她降落时，当地时钟显示的时间是 2012 年 3 月 11 日凌晨 3 点 10 分。那么请问这段旅程共花了多长时间？要回答这个问题有一个关键点，那就是在这段飞行的过程中发生了夏令时的转换，也就是说时间向前跃迁了。如果使用了带时区信息的时间戳，算出来的时间间隔就是 20 分钟，对于一段仅仅跨越旧金山海湾的短途飞行来说，这个答案显然是可信的。如果我们不使用带时区信息的数据类型，一定会得到错误的答案。

```
SELECT '2012-03-11 3:10 AM America/Los_Angeles'::timestamptz
 - '2012-03-11 1:50 AM America/Los_Angeles'::timestamptz;
```

以上查询得到的答案是 20 分钟，然而以下查询得到的答案却是 1 小时 20 分钟。

```
SELECT '2012-03-11 3:10 AM'::timestamp- '2012-03-11 1:50 AM'::timestamp;
```

我们再举几个例子来把这个问题讲得更透彻一些。如示例 5-10 所示，我输入时使用的是带时区的洛杉矶本地时间，但由于数据库服务器位于波士顿，所以查询时输出的时间是带时区信息的波士顿本地时间。请注意，输出显示附带了时差，这是没问题的，与我原始录入的时间之间仅仅是显示差异而已，在数据库系统内部是以 UTC 标准时间存储的。

示例 5-10　输入时使用的是一个时区的本地时间，输出却是另一个时区的本地时间

```
SELECT '2012-02-28 10:00 PM America/Los_Angeles'::timestamptz;

2012-02-29 01:00:00-05
```

在示例 5-11 中，我们要求返回的是不带时区的时间戳。因此这个查询在全世界任何地方的数据库服务器上执行都会返回相同的结果。

示例 5-11　将带时区信息的时间戳数据转换为不带时区的时间戳数据

```
SELECT '2012-02-28 10:00 PM America/Los_Angeles'::timestamptz
AT TIME ZONE 'Europe/Paris';

2012-02-29 07:00:00
```

以上查询其实回答了这么一个问题：洛杉矶本地时间 2012-02-28 10:00 p.m. 对巴黎来说是当地时间几点？请注意，查询结果是不带相对于 UTC 标准时间的时差的。另外请注意，可以通过官方名称而非 UTC 时差来指定一个时区，可以访问维基百科来查看所有时区的官方名称（http://en.wikipedia.org/wiki/Tz_database）。

5.3.2　日期时间类型的运算符和函数

时间间隔（`interval`）类型的引入极大简化了 PostgreSQL 中日期和时间类型的数学运算过程。如果没有 `interval` 类型，我们就得专门创建一堆函数来实现这些运算功能，很多其他数据库就是这么干的。通过 `interval` 类型，可以使用我们很熟悉的加减运算符对日期和时

间进行相加或者相减操作。下面的例子展示了可用于日期和时间类型的运算符和函数。

相加运算符（+）可以在一个时间类型值上加上一段时间间隔：

```
SELECT '2012-02-10 11:00 PM'::timestamp + interval '1 hour';

2012-02-11 00:00:00
```

你也可以将两个 interval 类型直接相加：

```
SELECT '23 hours 20 minutes'::interval + '1 hour'::interval;

24:20:00
```

相减运算符（-）可以从一个时间类型值中减去一段时间间隔：

```
SELECT '2012-02-10 11:00 PM'::timestamptz - interval '1 hour';

2012-02-10 22:00:00-05
```

示例 5-12 中展示了区间重叠运算符 OVERLAPS 的用法，如果两个参与运算的时间段有重叠，那么判定结果就是 true。这是 ANSI SQL 标准中规定的运算符，其效果等价于 overlaps 函数。OVERLAPS 谓词运算符需要四个形参，前两个是第一个时间段的首尾时间点，后两个是第二个时间段的首尾时间点。OVERLAPS 运算符会将这两个时间段看作半开半闭区间，也就是说起始时点包含在时段内，结束时点不包含在时段内。这与 BETWEEN 谓词运算符的逻辑是不一样的，BETWEEN 会认为起始点和结束点都是包含在区间内的。只要你设置的时间段的起始时点和结束时点不相同（如果相同的话就意味着时间段长度为 0，也就是说时间段变成了一个时间点），这个差异就不会造成什么问题。如果你经常需要使用 OVERLAPS 运算符，请务必注意这一点。

示例 5-12　对时间戳和日期类型使用 OVERLAPS 运算符

```
SELECT
      ('2012-10-25 10:00 AM'::timestamp, '2012-10-25 2:00 PM'::timestamp)
      OVERLAPS
      ('2012-10-25 11:00 AM'::timestamp,'2012-10-26 2:00 PM'::timestamp) AS x,
      ('2012-10-25'::date,'2012-10-26'::date)
      OVERLAPS
      ('2012-10-26'::date,'2012-10-27'::date) As y;

x  |y
---+---
t  |f
```

除了普通运算符和谓词运算符以外，PostgreSQL 还支持一些时间类型的函数。你可以从 PostgreSQL 官方手册的“日期和时间类型的相关函数和操作”一节中查到完整的函数列表，我们在此仅演示一个例子。

我们再次用到了用途广泛的 generate_series 函数。你可以对日期时间类型使用此函数，此时应使用 interval 类型值作为步长。

如示例 5-13 所示，可以用本地日期时间格式来输入日期，也可以使用在国际上更为通用的 ISO 格式“yyyy-mm-dd”来输入日期。PostgreSQL 会自动识别不同的输入格式。为保

险起见，我们倾向于使用 ISO 标准格式，因为在不同文化中日期的惯用格式是不一样的。由于本地设置的差异，在数据库服务器之间甚至是数据库实例之间也会存在这种日期格式差异。

实例 5-13　使用 `generate_series()` 函数来生成时间序列数组

```
SELECT (dt - interval '1 day')::date As eom
FROM generate_series('2/1/2012', '6/30/2012', interval '1 month') As dt;

eom
------------
2012-01-31
2012-02-29
2012-03-31
2012-04-30
2012-05-31
```

另一种经常使用的操作就是从日期和时间类型的数值中抽取出一部分。在 PostgreSQL 中，联用 `date_part` 和 `to_char` 函数可以实现此目标。示例 5-14 中除了演示这两个函数的用法外，还演示了带时区信息的日期时间类型在发生夏令时变换时的转换逻辑，为此我们特地挑选了一个美国东部时区（`US/East`）中横跨夏令时变化点的时间段。夏令时从凌晨 2 点生效，因此该表的最后一行就是夏令时变化以后的新时间。

示例 5-14　从日期时间类型中提取部分元素

```
SELECT dt, date_part('hour',dt) As hr, to_char(dt,'HH12:MI AM') As mn
FROM
generate_series(
        '2012-03-11 12:30 AM',
        '2012-03-11 3:00 AM',
        interval '15 minutes'
) As dt;

dt                     | hr | mn
-----------------------+----+----------
2012-03-11 00:30:00-05 |  0 | 12:30 AM
2012-03-11 00:45:00-05 |  0 | 12:45 AM
2012-03-11 01:00:00-05 |  1 | 01:00 AM
2012-03-11 01:15:00-05 |  1 | 01:15 AM
2012-03-11 01:30:00-05 |  1 | 01:30 AM
2012-03-11 01:45:00-05 |  1 | 01:45 AM
2012-03-11 03:00:00-04 |  3 | 03:00 AM
```

`generate_series` 函数默认生成的是 `timesatamptz` 类型的数据，需要显式转换为 `timestamp` 类型。

5.4　数组类型

数组在 PostgreSQL 中扮演着重要的角色。它在构造聚集函数、形成 `IN` 和 `ANY` 子句、承载数据类型转换过程中生成的中间值等领域发挥着重要作用。在 PostgreSQL 中，每种数据类型都有相应的以其为基础的数组类型。如果你自定义了一个数据类型，那么 PostgreSQL 会在后台自动为此类型创建一个数组类型。例如，`integer` 有一个相应的整数数组类型

integer[]，character 也有相应的字符数组类型 character[]，以此类推。下面将展示一些可以快速构造出数组的函数，可以免除你一个个元素录入的麻烦。此外还有一些用于管理数组的函数。你可以从 PostgreSQL 官方手册的"数组函数和运算符"一节中查到全部的数组函数和运算符列表。

5.4.1　数组构造函数

最基本的构造数组的方法就是一个个元素手动录入，语法如下：

```
SELECT ARRAY[2001, 2002, 2003] As yrs;
```

如果数组元素存在于一个查询返回的结果集中，那么可以使用这个略复杂一些的构造函数 array() 来生成数组：

```
SELECT array(
SELECT DISTINCT date_part('year', log_ts)
FROM logs
ORDER BY date_part('year', log_ts)
);
```

尽管 array 函数仅能用于将单字段的查询结果集转换为数组，但你依然可以指定一个复合数据类型作为查询结果，这种情况下可以获得多列结果。5.8.6 节会演示该用法。

你可以把一个直接以字符串格式书写的数组转换为一个真正的数组，语法如下：

```
SELECT '{Alex,Sonia}'::text[] As name, '{46,43}'::smallint[] As age;

name         | age
-------------+--------
{Alex,Sonia} | {46,43}
```

你还可以用 string_to_array 函数将一个用固定分隔符分隔的字符串转换为数组，如示例 5-15 所示。

示例 5-15　将一个分隔符格式的字符串转换为数组

```
SELECT string_to_array('ca.ma.tx', '.') As estados;

estados
----------
{CA,MA,TX}
(1 row)
```

array_agg 是一种聚合函数，它可用于将一组任何类型的数据转换为数组，如示例 5-16 所示。

示例 5-16　array_agg 函数的使用

```
SELECT array_agg(log_ts ORDER BY log_ts) As x
FROM logs
WHERE log_ts BETWEEN '2011-01-01'::timestamptz AND '2011-01-15'::timestamptz;
x
------------------------------------------
{'2011-01-01', '2011-01-13', '2011-01-14'}
```

在 PostgreSQL 9.5 中，array_agg 函数新增了支持数组型数据作为入参这一能力。在此前

的版本中，如果你试图通过 array_agg 将一批数组聚合为二维数组，它会报错说不支持。array_agg 支持数组型入参这一能力使得根据一维数组构建出多维数组成为一件很容易的事情，如示例 5-17 所示。

示例 5-17　根据一维数组构建多维数组

```
SELECT array_agg(f.t)
FROM ( VALUES ('{Alex,Sonia}'::text[]),
    ('{46,43}'::text[] ) ) As f(t);

array_agg
----------------------
{{Alex,Sonia},{46,43}}
(1 row)
```

不过要想使用该功能，被聚合的基础数组中的元素类型必须相同，而且被聚合的基础数组的维度必须一样。为了保证这一点，在示例 5-17 中，我们对其中的年龄数组做了类型转换，统一为 text。另外也可以看到，上面例子中待聚合的基础数组中的元素个数都是相同的：两个人名，两个年龄。如果一批数组中每个数组的内部元素个数都相同，则这一批数组可以称为**平衡数组**。

5.4.2　将数组元素展开为记录行

另外一个常用的数组操作函数是 unnest，通过它可以将数组元素纵向展开成一个包含若干条记录的结果集，如示例 5-18 所示。

示例 5-18　使用 unnest 函数将数组纵向展开

```
SELECT unnest('{XOX,OXO,XOX}'::char(3)[]) As tic_tac_toe;

tic_tac_toe
---
XOX
OXO
XOX
```

你可以在一个 SELECT 语句中使用多个 unnest 函数，但如果每个 unnest 展开后的记录行数不一致，或者说“对不齐”，那么得到的最终结果将是这些结果集之间的笛卡儿积，看起来不太好理解。

示例 5-19 演示了一个 unnest 展开后可对齐的结果集，也就是说每个 unnest 都输出 3 行记录，最终连接成的记录也是 3 行。

示例 5-19 多个可对齐数组的展开效果

```
SELECT
unnest('{three,blind,mice}'::text[]) As t,
unnest('{1,2,3}'::smallint[]) As i;

t     |i
------+-
three |1
blind |2
mice  |3
```

如果你从上述一个数组中拿掉一个元素，那么两个数组的元素就无法对齐了，此时展开得到的结果如示例 5-20 所示。

示例 5-20　多个无法对齐的数组展开后的效果

```
SELECT
unnest( '{blind,mouse}'::varchar[]) As v,
unnest('{1,2,3}'::smallint[]) As i;

v     |i
------+-
blind |1
mouse |2
blind |3
mouse |1
blind |2
mouse |3
```

PostgreSQL 9.4 中，unnest 函数支持了多个入参，该函数会在数组不平衡的位置[1] 填入空值 null。新的 unnest 的主要缺点是，它只能用在 FROM 子句中。示例 5-21 使用了刚刚介绍过的 PostgreSQL 9.4 新支持的 unnest，重新展开了示例 5-20 中展开过的不平衡数组。

示例 5-21 使用支持多入参的 unnest 展开不平衡数组

```
SELECT * FROM unnest('{blind,mouse}'::text[], '{1,2,3}'::int[]) AS f(t,i);

t      | i
-------+---
blind  | 1
mouse  | 2
<NULL> | 3
```

5.4.3　数组的拆分与连接

PostgreSQL 支持使用 start:end 语法对数组进行拆分。操作结果是原数组的一个子数组。例如，如果要得到一个仅包含当前数组第 2 个至第 4 个元素的新数组，可以使用以下语法：

```
SELECT fact_subcats[2:4] FROM census.lu_fact_types;
```

如果要将两个数组连接到一起，可以使用连接运算符 ||：

```
SELECT fact_subcats[1:2] || fact_subcats[3:4] FROM census.lu_fact_types;
```

可以通过下面的语法来为一个现有数组添加元素：

```
SELECT '{1,2,3}'::integer[] || 4 || 5;
```

得到的结果是 {1,2,3,4,5}。

注 1：如果某个数组相对其他数组来说元素个数少，则缺少的那些元素的位置就是不平衡的位置。
——译者注

5.4.4 引用数组中的元素

一般来说，我们会通过数组下标来引用数组元素，请特别注意 PostgreSQL 的数组下标从 1 开始。如果你试图越界访问一个数组，也就是说数组下标已经超过了数组元素的个数，那么不会返回错误，而是会得到一个空值 NULL。下面的例子演示了获取数组的第一个和最后一个元素的方法：

```
SELECT
    fact_subcats[1] AS primero,
    fact_subcats[array_upper(fact_subcats, 1)] As segundo
FROM census.lu_fact_types;
```

我们使用 array_upper 函数来获取数组元素的个数，该函数的第二个必选入参表示数组的维度。在本例中，数组是一维的，但 PostgreSQL 支持多维数组。

5.4.5 数组包含性检查

PostgreSQL 支持多种数组类型的运算符。前面已经见过了连接运算符（||），它可以将多个数组合并为一个，也可以为现有数组添加新元素，详见 5.4.3 节。此外，数组类型的运算符还有 =、<>、<、>、@>、<@ 以及 &&。这些运算符要求两边数组的数据类型相同。如果你在数组类型的字段上建立了 GiST 或者 GIN 索引，那么这些运算符可以用上索引。

重叠判定运算符（&&）的作用是：如果两个数组有任何共同的元素，则返回 true；否则返回 false。示例 5-22 将查出我们表中所有满足“fact_subcats 数组字段中含有 OCCUPANCY STATUS 或者是 For rent 这两个值中的一个”这一条件的记录。

示例 5-22 数组重叠判定运算符

```
SELECT fact_subcats
FROM census.lu_fact_types
WHERE fact_subcats && '{OCCUPANCY STATUS,For rent}'::varchar[];

fact_subcats
-----------------------------------------------------------
{S01,"OCCUPANCY STATUS","Total housing units"...}
{S02,"OCCUPANCY STATUS","Total housing units"...}
{S03,"OCCUPANCY STATUS","Total housing units"...}
{S10,"VACANCY STATUS","Vacant housing units","For rent"...}
(4 rows)
```

只有当两个数组中的所有元素及其排列顺序都完全相同时，等值判定运算符（=）才会返回 true。如果你并不关心两个数组的元素顺序是否完全相同，只是想知道一个数组中的元素集合是否是另一个数组的子集，可以使用包含关系判定运算符（@>、<@）。示例 5-23 中演示了包含（@>）和被包含（<@）这两个运算符的区别。

示例 5-23 数组包含关系判定运算符

```
SELECT '{1,2,3}'::int[] @> '{3,2}'::int[] AS contains;

contains
--------
t
```

```
(1 row)
SELECT '{1,2,3}'::int[] <@ '{3,2}'::int[] AS contained_by;

contained_by
------------
f
(1 row)
```

5.5 区间类型

区间数据类型可以表达一个带有起始值和结束值的值区间。PostgreSQL 为区间类型提供了很多配套的运算符和函数，例如判定区间是否重叠，判定某个值是否落在区间内，以及将相邻的若干区间合并为一个完整的区间等。在出现区间类型之前，类似操作只能通过写函数实现，这种操作很繁琐，不仅低效而且很容易出错，并且写出的函数不一定能达到预想的效果，在对于时间类型的操作中尤其如此。在我们自己的项目中，我们在所有需要表示时间范围的表中都用上了区间类型，事实证明效果很好。我们希望你也能分享我们这一成功经验。

有了区间类型后，就不再需要用两个字段来定义一个区间。假设我们希望定义一个大于等于 −2 小于 2 的整数区间，该区间的写法是 `[-2,2)`，左边中括号表示左边是闭区间，即值域包含 −2；右边小括号表示右边是开区间，即值域不包含 2。那么 `[-2,2)` 这个整数区间包含的元素有：−2，−1，0，1。类似地，可以知道以下整数区间所包含的元素。

- 整数区间 `(-2,2]` 含四个元素：`-1`、`0`、`1`、`2`。
- 整数区间 `(-2,2)` 含三个元素：`-1`、`0`、`1`。
- 整数区间 `[-2,2]` 含五个元素：`-2`、`-1`、`0`、`1`、`2`。

5.5.1 离散区间和连续区间

PostgreSQL 对离散区间和连续区间是区别对待的。整数类型或者日期类型的区间是离散区间，因为区间内每一个值都是可以被枚举出来的。数字区间或者时间戳区间就是一个连续区间，因为区间内的值有无限多。

一个离散区间有多种表示方法，比如前面提到的 `[-2,2)` 这个例子，它就可以换用多种写法而且每种方式的效果完全一样：`[-2,1]`、`(-3,1]`、`(-3,2)` 和 `[-2,2)`。这四种写法中，PostgreSQL 规定 `[-2,2)` 为规范写法，并不是因为这种写法有什么优势，仅仅是因为统一后有利于运算，不用每次计算时都得先考虑是开区间还是闭区间这件事。PostgreSQL 会自动对所有的离散区间进行规范化，不管是存储还是显示时都会这么做。因此，如果你输入了一个时间区间 `(2014-1-5,2014-2-1]`，那么 PostgreSQL 会自动把它改写为 `[2014-01-06,2014-02-02)`。

5.5.2 原生支持的区间类型

PostgreSQL 原生支持六种区间类型，都是关于数字型和日期时间型的。

`int4range`、`int8range`

　这是整数型离散区间，其定义符合前闭后开的规范化要求。

numrange
　　这是连续区间，可以用于描述小数、浮点数或者双精度数字的区间。

daterange
　　这是不带时区信息的日期离散区间。

tsrange、tstzrange
　　这是时间戳（日期加时间）类型的连续区间，秒值部分支持小数。tsrange 不带时区信息，tstzrange 带时区信息。

对于数字类型的区间来说，如果区间的起点值或者终点值未指定，那么 PostgreSQL 会自动为其填入 null 值。理论上讲，你可以将该 null 解释为代表左侧的 -infinity（负无穷）或右侧的 infinity（正无穷）。实际上，你会受限于特定数据类型的最小值和最大值。比如对于 int4range 数据类型来说，区间 (,) 实际上代表的是 [-2147483648,2147483647)。

对于时间类型的区间来说，-infinity 和 infinity 就是有效的上限和下限。

除了系统原生支持的区间类型外，你还可以自定义区间类型，可以设定为离散区间也可以设定为连续区间。

5.5.3　定义区间的方法

任何类型的区间都是由相同数据类型的起点值和终点值外加表示区间开闭的符号 [、]、(、) 构成的，如示例 5-24 所示。

示例 5-24　使用类型转换的方法来定义区间

```
SELECT '[2013-01-05,2013-08-13]'::daterange; ❶
SELECT '(2013-01-05,2013-08-13]'::daterange; ❷
SELECT '(0,)'::int8range; ❸
SELECT '(2013-01-05 10:00,2013-08-13 14:00]'::tsrange; ❹

[2013-01-05,2013-08-14)
[2013-01-06,2013-08-14)
[1,)
("2013-01-05 10:00:00","2013-08-13 14:00:00"]
```

❶ 定义了一个从 2013-01-05 到 2013-08-13 的日期型闭区间。请注意此处区间终点的写法是不规范的。

❷ 定义了一个从 2013-01-05 到 2013-08-13 的日期型半开半闭区间。请注意此处区间起点和终点的写法都是不规范的。

❸ 定义了一个大于 0 的整数区间。请注意此处区间起点的写法是不规范的。

❹ 定义了一个从 2013-01-05 10:00 AM 到 2013-08-13 2 PM 的半开半闭连续区间。



PostgreSQL 中的日期时间类型可以用 -infinity 表示负无穷，用 infinity 表示正无穷。为了与传统写法保持一致，建议以约定俗成的“前闭后开”方式来书写时间范围，即区间起点使用左中括号“[”，区间终点使用右小括号“)”，比如 [-infinity, intifinity)。

区间也可以使用区间构造函数来定义，该构造函数的名称与区间类型名称是一致的，可以输入两个或者三个参数。示例如下：

```
SELECT  daterange('2013-01-05','infinity','[]');
```

第三个参数是区间边界开闭标识符，如果不填则默认为前开后闭 [)。为清晰起见，建议你总是显式指定该参数，因为其默认值不是那种非常显而易见的值，容易被记错。

5.5.4 定义含区间类型字段的表

时间类型区间是很常用的，假设你有一张 employment 表，表中存储了公司聘请雇员的历史记录。你可以像示例 5-25 那样用时间区间来定义一个员工在公司的服务年限，而不需要用起始时间和结束时间两个单独的字段来表示。在示例 5-25 中，我们给 period 列添加了一个索引，以使用我们的区间列加速查询。

示例 5-25 建立一个带有日期区间类型字段的表

```
CREATE TABLE employment (id serial PRIMARY KEY, employee varchar(20),
period daterange);
CREATE INDEX ix_employment_period ON employment USING gist (period);❶
INSERT INTO employment (employee,period)
VALUES
        ('Alex','[2012-04-24, infinity)'::daterange),
        ('Sonia','[2011-04-24, 2012-06-01)'::daterange),
        ('Leo','[2012-06-20, 2013-04-20)'::daterange),
        ('Regina','[2012-06-20, 2013-04-20)'::daterange);
```

❶ 在区间字段上建立一个 GiST 索引。

5.5.5 适用于区间类型的运算符

区间类型上用得最多的两个运算符是重叠运算符（&&）和包含运算符（@>）。接下来将对这两个运算符进行介绍。要了解区间运算符的完整列表，请参考 PostgreSQL 官方手册中的“区间类型运算符”一节。

1. 重叠运算符

顾名思义，重叠判定运算符 && 的作用就是判定两个区间是否有重叠部分，如果有则返回 true，否则返回 false。示例 5-26 演示了该运算符的用法，其中还使用了 string_agg 函数将雇员名单列表合并成一个文本字段。

示例 5-26 查询谁与谁曾经同时在公司工作过

```
SELECT
        e1.employee,
        string_agg(DISTINCT e2.employee, ', ' ORDER BY e2.employee) As colleagues
FROM employment As e1 INNER JOIN employment As e2
ON e1.period && e2.period
WHERE e1.employee <> e2.employee
GROUP BY e1.employee;

employee | colleagues
---------+------------------
```

```
Alex     | Leo, Regina, Sonia
Leo      | Alex, Regina
Regina   | Alex, Leo
Sonia    | Alex
```

2. 包含与被包含关系运算符

对于包含关系运算符 `@>` 来说，第一个参数是区间，第二个参数是待判定的值。如果第二个参数的值落在第一个参数的区间内，运算符就返回 `true`，否则返回 `false`。示例 5-27 演示了其用法。

示例 5-27　查询当前还在公司工作的雇员名单

```
SELECT employee FROM employment WHERE period @> CURRENT_DATE GROUP BY employee;

employee
----
Alex
```

`<@` 是用于判定被包含关系是否成立的运算符，它的第一个参数是待判定的值，第二个参数是区间，其用法与包含关系运算符完全一致，不再赘述。

5.6　JSON数据类型

PostgreSQL 支持 JSON 数据类型并提供了很多相关的函数。JSON 已成为 Web 开发领域最流行的数据传递格式。PostgreSQL 9.3 中针对 JSON 类型做了显著的功能增强，新增了一些用于实现子树抽取、内容编辑以及与其他数据类型进行互转的操作函数。PostgreSQL 9.4 中引入了 JSONB 数据类型，该类型是 JSON 类型的二进制版本，它与 JSON 类型最主要的差别是 JSONB 可以支持索引而 JSON 不能（只能使用非常受限制的函数索引）。PostgreSQL 9.5 中为 JSONB 类型引入了更多的支持函数，包括可用于修改 `jsonb` 对象中的子元素的函数。PostgreSQL 9.6 中引入了 `josnb_insert` 函数，可以实现往一个现有的 `jsonb` 对象中插入数据，即支持往 JSONB 内部的数组对象中增加新元素，也可以实现在 JSONB 内部增加一个新的键值对。

5.6.1　插入JSON数据

要想在表中存储 JSON 数据，只需建一个 `json` 类型的字段即可，语法如下：

```
CREATE TABLE persons (id serial PRIMARY KEY, person json);
```

示例 5-28 的语句向表中插入了一条 JSON 记录。PostgreSQL 会自动对插入的 JSON 文本进行格式检查以确保其格式合法。请注意，无法将无效的 JSON 字符串存储到某个 JSON 列中，而且也没有什么办法可以把无效的 JSON 字符串转换为 JSON 类型。

示例 5-28　插入一个 JSON 字段

```
INSERT INTO persons (person)
VALUES (
    '{
        "name":"Sonia",
        "spouse":
```

```
            {
                "name":"Alex",
                "parents":
                {
                    "father":"Rafael",
                    "mother":"Ofelia"
                },
                "phones":
                [
                    {
                        "type":"work",
                        "number":"619-722-6719"
                    },
                    {
                        "type":"cell",
                        "number":"619-852-5083"
                     }
                 ]
            },
            "children":
            [
                {
                    "name":"Brandon",
                    "gender":"M"
                },
                {
                    "name":"Azaleah",
                    "girl":true,
                    "phones": []
                }
            ]
        }'
    );
```

5.6.2 查询JSON数据

要想对 JSON 数据的层次化结构进行访问，最简单的方法就是使用路径指向符。示例 5-29 演示了一些常见的用法。

示例 5-29　查询 JSON 字段

```
SELECT person->'name' FROM persons;
SELECT person->'spouse'->'parents'->'father' FROM persons;
```

也可以通过路径数组的形式进行查询，如下例所示：

```
SELECT person#>array['spouse','parents','father'] FROM persons;
```

注意，如果查询语句中使用了路径数组，前面必须用 `#>` 指向符。

可以通过指定下标来访问 JSON 内部数组的某个特定元素。请注意，JSON 数组的起始元素下标与 PostgreSQL 数组类型不同，前者从 0 开始，后者从 1 开始。

```
SELECT person->'children'->0->'name' FROM persons;
```

上面这句话用路径数组语法来表达就是：

```
SELECT person#>array['children','0','name'] FROM persons;
```

上面的这些例子中，查询返回的数据类型都是 JSON 内部元素的基础类型（数字、字符串、布尔值）。如果希望返回的值都统一转换为文本型，那么只需在路径指向符中再增加一个 > 符即可：

```
SELECT person->'spouse'->'parents'->>'father' FROM persons;
SELECT person#>>array['children','0','name'] FROM persons;
```

可以看到，如果联用多个 -> 指向符，只需把最后一个指向符改写为 ->> 即可。

json_array_elements 函数的入参是一个 JSON 数组，该函数会将该 JSON 数组中的每个元素拆成单独一行输出，如示例 5-30 所示。

示例 5-30　用 json_array_elements 函数展开 JSON 数组

```
SELECT json_array_elements(person->'children')->>'name' As name FROM persons;

name
-------
Brandon
Azaleah
(2 rows)
```



JSON 类型本质上表达了层次化结构的树型数据，当需要访问树上的元素时，强烈建议你使用指向符语法。该语法非常简洁，而且对 JSONB 类型也适用（稍后将介绍 JSONB 类型）。PostgreSQL 提供了一个功能与指向符语法对等的函数 json_extract_path，该函数可以接受不定长入参（也就是说该函数的入参个数可以无限多）。第一个参数是待访问的 JSON 对象，后续参数就是层次化访问路径上每一层的 key 值。->> 运算符也有一个功能完全对等的函数，名为 json_extract_path_text，其用法不再赘述。

5.6.3　输出JSON数据

PostgreSQL 除了可以查询库中已有的 JSON 数据外，还支持将别的数据类型转换为 JSON 类型。接下来的例子将演示系统内置的 JSON 转换函数的用法，这类函数可以将其他数据类型转换为 JSON 类型。

示例 5-31 中将使用 row_to_json 函数将示例 5-28 中导入的数据的部分字段转换为 JSON 数据。

示例 5-31　将多条记录转换为单个 JSON 对象（PostgreSQL 9.3 及之后的版本才支持该语句）

```
SELECT row_to_json(f) As x
FROM (
    SELECT id, json_array_elements(person->'children')->>'name' As cname FROM persons
) As f;

                x
--------------------------
```

```
{"id":1,"cname":"Brandon"}
{"id":1,"cname":"Azaleah"}
(2 rows)
```

如果要将 persons 表中的所有记录行整体打包转换为一个 JSON 对象，可以使用以下语法：

```
SELECT row_to_json(f) As jsoned_row FROM persons As f;
```

“查询时将一行记录作为单个字段输出”这种功能只有 PostgreSQL 才支持。该功能对于创建复合 JSON 对象特别有用。我们将在 7.2.12 节中深入讨论此特性，并且将在示例 7-20 中演示如何使用 array_agg 和 array_to_json 函数将多条记录转换为一个 JSON 对象后输出。9.3 版新增了对 json_agg 函数的支持，示例 7-21 中将演示该函数的用法。

5.6.4　JSON类型的二进制版本：jsonb

PostgreSQL 9.4 中引入了新的 jsonb 数据类型。该类型使用的运算符与 json 类型完全相同；该类型的处理函数与 json 类型的处理函数一一对应，仅在命名上略有差别（前者以“jsonb”开头，后者以“json”开头），另外 jsonb 类型还比 json 类型多了一些新的函数。由于 jsonb 类型的数据在存入库中时经过了预解析，因此在处理过程中无须再次进行文本解析，所以其处理性能远超 json 类型。jsonb 数据类型和 json 数据类型的关键区别如下所示。

- json 是以原始文本格式存储的，而 jsonb 存储的是原始文本解析以后生成的二进制数据结构，该二进制结构中不再保存原始文本中的空格，存储下来的数字的形式也发生了一定的变化，并且对其内部记录属性值进行了排序。例如，文本中的 e-5 这种数字会被转换为对应的小数存储。
- jsonb 不允许其内部记录的键值重复，如果出现重复则会从中自动选择一条，其余的重复记录会被丢弃，但 json 类型中记录键值重复是允许的。Michael Paquier 的博文“Manipulating jsonb data by abusing of key uniqueness”中演示了若干例子。
- jsonb 类型的字段上可以直接建立 GIN 索引（该类索引在 6.3 节中有相关介绍），但 json 类型字段上却只能建立函数索引，因为只有通过函数才能从 JSON 的字符串中提取出具体字段值。

为了说明以上概念，我们另外新建一张与前面 persons 结构类似的 persons 表，只不过这次用的是 jsonb 类型：

```
CREATE TABLE persons_b (id serial PRIMARY KEY, person jsonb);
```

重复执行示例 5-28 的步骤，往新表中插入记录。

目前为止还没体现出 JSON 和 JSONB 的差别，但在对两张表分别执行查询时就能看出来了。为了让 JSONB 类型的二进制字段值能够显示，PostgreSQL 会自动将其转换为规范化的文本表示形式，如示例 5-32 所示。

示例 5-32　JSONB 与 JSON 类型输出格式对比

```
SELECT person As b FROM persons_b WHERE id = 1; ❶
SELECT person As j FROM persons WHERE id = 1; ❷
b
```

```
-----------------------------------------------------------------------------
{"name": "Sonia",
"spouse": {"name": "Alex", "phones": [{"type": "work", "number": "619-722-6719"},
{"type": "cell", "number": "619-852-5083"}],
"parents": {"father": "Rafael", "mother": "Ofelia"}},
"children": [{"name": "Brandon", "gender": "M"},
{"girl": true, "name": "Azaleah", "phones": []}]}
(1 row)

                      j
---------------------------------------------
{
    "name":"Sonia",
         "spouse":
         {
             "name":"Alex",
             "parents":
             {
                 "father":"Rafael",
                 "mother":"Ofelia"
              },
              "phones":
              [
                  {
                      "type":"work",
                      "number":"619-722-6719"+
                  },
                  {
                      "type":"cell",
                      "number":"619-852-5083"+
                  }
              ]
          },
          "children":
          [
              {
                  "name":"Brandon",
                  "gender":"M"
              },
              {
                  "name":"Azaleah",
                  "girl":true,
                  "phones": []
              }
          ]
     }
(1 row)
```

❶ 可以看出，jsonb 类型的输出是对输入的内容进行了重新格式化，并删掉了输入时文本中的空白。此外，插入记录时属性字段的顺序信息是不保留的。

❷ json 类型的输出保持了输入时的原样，包括原文本中的空白以及属性字段的顺序。

jsonb 与 json 的处理函数一一对应，但函数名略有不同，而且 jsonb 比 json 多了一些处理函数。例如，json 适用的 json_extract_path_text 和 json_each 函数对应于 jsonb 适用

的 jsonb_extract_path_text 和 jsonb_each 函数。二者的运算符完全相同，因此如果想把 5.6.2 节的示例改造为适用 jsonb 类型，只需把表名替换一下，然后把 json_array_elements 替换为 jsonb_array_elements 即可。

jsonb 比 json 多支持的运算符有以下几个：等值判定运算符（=）、包含关系判定运算符（@>）、被包含关系判定运算符（<@）、键值存在判定运算符（?）、判定一组键值中是否有任意一个已存在的运算符（?|），以及判定一组键值中的每一个是否均已存在的运算符（?&）。

假设要列出所有孩子姓名为"Brandon"的人员名单，就可以使用包含关系判定运算符，如示例 5-33 所示。

示例 5-33　JSONB 包含关系运算符的使用

```
SELECT person->>'name' As name
FROM persons_b
WHERE person @> '{"children":[{"name":"Brandon"}]}';

name
-----
Sonia
```

如果在 jsonb 列上创建了 GIN 索引，那么前述这几个运算符的操作速度是极快的：

```
CREATE INDEX ix_persons_jb_person_gin ON persons_b USING gin (person);
```

我们演示用的这些表都很小，因此规划器可能会选择走全表扫描而不是走索引查询，但如果有更多的记录，示例 5-33 中这种语句是一定会用上索引的。

5.6.5　编辑JSONB类型的数据

PostgreSQL 9.5 中为了支持对 JSONB 类型的数据进行修改，引入了原生的 jsonb 连接运算符（||）以及删减运算符（-、#-），同时还引入了一些辅助操作函数。注意，json 类型并不支持这些运算符。如果要在 PostgreSQL 9.5 之前的版本中实现相同的功能，必须通过 JavaScript 扩展语言来实现，可参考 8.5 节的内容。

连接运算符可用于对一个 jsonb 对象新增或者替换其内部属性字段。在示例 5-34 中，我们为 Gomez 家对应的 jsonb 对象新增了一个地址属性，并使用 RETURNING 语法返回了更新后字段的值。关于 RETURNING 语法，请参考 7.2.10 节的内容。返回的新值中会含有地址属性字段。

示例 5-34　使用 JSONB 的 || 运算符来增加地址

```
UPDATE persons_b
SET person = person || '{"address": "Somewhere in San Diego, CA"}'::jsonb
WHERE person @> '{"name":"Sonia"}'
RETURNING person;
profile
---------------------------------------------------------------------------
{"name": "Sonia", ... "address": "Somewhere in San Diego, CA", "children": ...}
(1 row)
UPDATE 1
```

由于 JSONB 类型要求内部属性字段的键值必须唯一，因此如果你试图插入一个重复名称的属性字段，原来的同名属性字段值会被替换掉。因此如需修改旧地址，可以使用与示

例 5-34 中完全相同的语句，只需把其中的“Somewhere in San Diego, CA”替换为新的地址即可。

如果需要把地址属性删掉，可以使用 - 运算符，如示例 5-35 所示。

示例 5-35　使用 JSONB 的 - 运算符来移除某个属性

```
UPDATE persons_b
SET person = person - 'address'
WHERE person @> '{"name":"Sonia"}';
```

请注意，简单地使用 - 运算符只能删掉 JSONB 层次化树结构上第一层的元素。如果需要删除内部某一层的特定属性怎么办呢？此时可以使用 #- 运算符。该运算符使用一个表达了待删除属性所在路径的文本值数组作为入参。在示例 5-36 中，我们删除了 Azaleah 的 `girl` 属性。

示例 5-36　使用 JSONB 的 #- 运算符删除内嵌的某个元素

```
UPDATE persons_b
SET person = person #- '{children,1,girl}'::text[]
WHERE person @> '{"name":"Sonia"}'
RETURNING person->'children'->1;

{"name": "Azaleah", "phones": []}
```

如需删除 JSONB 内部数组的某个特定元素，请指明其下标。由于 JavaScript 语言中数组元素的下标从 0 开始计数，所以如果要删除第二个子节点的某个元素，下标需写成 1 而不是 2。如果需删除整个名为“Azalean”的子节点，可以写成 `'{children,1}'::text[]`。

如需插入一个 `gender` 属性或者是修改 `gender` 属性值，可以使用 `jsonb_set` 函数，如示例 5-37 所示。

示例 5-37　使用 `jsonb_set` 函数修改内部嵌套的属性值

```
UPDATE persons_b
SET person = jsonb_set(person,'{children,1,gender}'::text[],'"F"'::jsonb, true)
WHERE person @> '{"name":"Sonia"}';
```

`jsonb_set` 函数有四个入参，其定义的形式为：`jsonb_set(jsonb_to_update, text_array_path, new_jsonb_value, allow_creation)`。如果将 `allow_creation` 置为 `false`，当所需修改的属性值不存在时，该函数会返回错误。

5.7　XML数据类型

XML 和 JSON 这两种数据类型都属于非规范化数据，在关系型数据库中存储这类数据其实是有争议的。然而，所有的高级关系型数据库（比如 IBM DB2、Oracle、SQL Server）中都支持 XML 数据类型。作为最先进的开源关系型数据库，PostgreSQL 自然也会支持 XML 数据类型，并且还提供了大量 XML 操作函数。我们发表过很多关于在 PostgreSQL 中使用 XML 数据类型的技术文章。PostgreSQL 原生支持创建、管理和解析 XML 数据的函数，详细列表可参见 PostgreSQL 官方手册中“XML 函数”一节。与 `jsonb` 数据类型不一样，目前没有哪种索引类型支持直接对 XML 数据类型进行索引，因此只能使用函数索引对其一部分数据进行索引，这一点与 `json` 是相同的。

5.7.1 插入XML数据

在往一个 xml 数据类型的列中插入数据时，PostgreSQL 会自动判定并确保只有格式合法的 XML 才会创建成功。text 类型字段中也可以存入一段 XML 文本，但是存入时不会进行格式合法性判断，这一点是 text 与 xml 类型的区别。不过请注意，即使 XML 文本的内容中附带了 DTD 或者 XSD 的格式描述，PostgreSQL 也不会按照这些格式要求来对 XML 的格式进行验证。为了梳理一下构成有效 XML 的要素，示例 5-38 展示了通过将某个列声明为 xml 并照常将数据插入到该列中，如何将 XML 数据追加到表中。

示例 5-38　插入 XML 字段记录

```
CREATE TABLE families (id serial PRIMARYKEY, profile xml);
INSERT INTO families(profile)
VALUES(
    '<family name="Gomez">
        <member><relation>padre</relation><name>Alex</name></member>
        <member><relation>madre</relation><name>Sonia</name></member>
        <member><relation>hijo</relation><name>Brandon</name></member>
        <member><relation>hija</relation><name>Azaleah</name></member>
        </family>');
```

XML 数据的格式是千变万化的，你可以为 XML 字段设置一个 check 约束以确保输入的 XML 数据都符合某种格式（如需了解 check 约束的详细信息，请参考 6.2.3 节的内容）。示例 5-39 中创建了一个 check 约束，该约束要求输入的 XML 数据中的 family 节点下都有一个 relation 节点。'/family/member/relation' 是 XPath 语法，XPath 是一种能够在 XML 树状结构中定位到指定元素的语法。

示例 5-39　确保所有 XML 字段记录中都有至少一个 member 节点和一个 relation 节点

```
ALTER TABLE families ADD CONSTRAINT chk_has_relation
CHECK (xpath_exists('/family/member/relation', profile));
```

如果试图插入这样一条记录：

```
INSERT INTO families (profile) VALUES ('<family name="HsuObe"></family>');
```

我们会看到这样的报错信息：ERROR: new row for relation "families" violates check constraint "chk_has_relation"（错误：试图插入“families”表中的新记录违反了约束“chk_has_relation”的要求）。

如果需要基于 DTD 或者 XSD 对 XML 数据进行格式检查，你需要自行编写格式检查函数，然后将此函数放到 check 约束中调用。PostgreSQL 目前还没有原生支持基于 DTD 或者 XSD 的格式检查。

5.7.2 查询XML数据

查询 XML 数据时，xpath 函数会发挥重要作用。该函数的第一个参数是一个 XPath 查询表达式，第二个参数是一个 xml 对象。查询结果是 XPath 查询语句所要查找的 XML 元素的列表。示例 5-40 中查询出了所有的家庭成员，查询中同时使用了 xpath 和 unnest 函数，其中 unnest 函数用于将数组转换成结果集。这样我们就把一段 XML 中的零碎信息提取出来并转换成了文本。

示例 5-40　查询 XML 字段

```
SELECT ordinality AS id, family,
    (xpath('/member/relation/text()', f))[1]::text As relation,
    (xpath('/member/name/text()', f))[1]::text As mem_name ❶
FROM (
    SELECT
        (xpath('/family/@name', profile))[1]::text As family, ❷
        f.ordinality, f.f
        FROM families, unnest(xpath('/family/member', profile)) WITH ORDINALITY AS f
) x;❸
 id | family | relation | mem_name
----+--------+----------+----------
  1 | Gomez  | padre    | Alex
  2 | Gomez  | madre    | Sonia
  3 | Gomez  | hijo     | Brandon
  4 | Gomez  | hija     | Azaleah
(4 rows)
```

❶ 获取每个 member 元素的 relation 标签和 name 标签中包含的文本元素。此处的语法中必须加数组下标，因为 xpath 语法返回的查询结果是数组类型的，即使返回的数组中只有一个元素也得加下标才能访问。

❷ 获取 family 根节点的 name 属性值。访问属性值的语法为 @attribute_name。

❸ 将 SELECT 语句的查询结果拆分为多个子元素，这些子元素包括 <member>、<relation>、</relation>、<name>、</name> 和 </member>。xpath 的斜杠语法表示要获取当前指定节点的子节点的内容。例如，xpath('/family/member', 'profile') 将以数组形式返回 profile 字段中 family 节点下所有 member 子节点的内容。xpath('/family/@name', 'profile') 返回的是 family 节点的 name 属性的值。默认情况下，xpath 返回的是包含前后标签部分的完整节点内容，加了 text() 以后，返回的就是该节点中包含的文本的内容。

PostgreSQL 10 中新增支持了 ANSI-SQL 中的 XMLTABLE 语法。XMLTABLE 可以基于预定义好的转换规则，将一段 XML 文本映射为独立的行和列。接下来使用 XMLTABLE 语法把示例 5-40 再实现一遍。

示例 5-41　使用 XMLTABLE 语法来查询 XML 数据

```
SELECT xt.*
  FROM families,
       XMLTABLE ('/family/member' PASSING profile  ❶
                 COLUMNS ❷
                    id FOR ORDINALITY , ❸
                    family text PATH '../@name' , ❹
                    relation text NOT NULL , ❺
                    member_name text PATH 'name' NOT NULL
                ) AS xt;

 id | family | relation | mem_name
----+--------+----------+----------
  1 | Gomez  | padre    | Alex
  2 | Gomez  | madre    | Sonia
  3 | Gomez  | hijo     | Brandon
  4 | Gomez  | hija     | Azaleah
(4 rows)
```

❶ XMLTABLE 定义中的第一部分是一个 XML 路径，表明从 XML 对象的哪个具体位置抽取数据行。PASSING 关键字后跟字段名，表示从哪个字段中抽取行数据，该字段一定要是 xml 类型。此处我们使用 families 表的 profile 字段。
❷ COLUMNS 关键字表示下面即将定义从 XML 数据中抽取出来的字段列表。
❸ 此前介绍过，在支持返回结果集的函数中可以使用 WITH ORDINALITY 语法来标记所返回记录的行号，此处可以类似地使用 FOR ORDINALITY 语法来标记查询结果的行号。
❹ 你可以使用 ../ 表达式定位到当前行位置的上一层。此处我们使用 ../@name 来获取 family 节点的 name 属性，该属性所在的层级比 family/member 节点要高一级。@ 符号表示要取的是一个属性值（XML 中属性值的语法形式为 name='a value'）而不是一个元素。
❺ 如果原始 XML 中待取数据的路径上的元素名与预定义的转换规则中指定的字段名相同，则无须在规则定义中为目标字段设定 PATH 值（即该字段的数据来源路径）。本例中，由于原始 XML 中 /family/member/ralation 这个路径上的 relation 元素名与转换规则中定义的 relation 字段同名，所以无须在这里写 PATH 来指定数据来源路径。

5.8 全文检索

我相信你一定曾经在某些网站上体验过关键词搜索功能。如果是电商类网站，那么可以搜索出匹配关键词的商品列表；如果是电影类网站，那么可以搜索出符合关键词过滤条件的电影；如果是知识问答类网站，那么可以搜索出匹配关键词的问题和答案。

要想实现通过关键词对文本内容进行搜索，可以使用常见的 like 或者 ilike（忽略大小写的 like）这样的匹配方法，也可以使用强大的正则表达式或者 soundex 语音匹配算法。但这些方法都存在一个问题：它们无法实现基于自然语言的匹配。例如，如果你想搜索 LGBT[2] 类的电影，把“LGBT”作为搜索关键词，那么那些电影介绍中带有“lesbian、gay、bisexual、transgendered”的电影就搜不出来，因为文本不匹配。

FTS（full text search，全文检索）是一个带有一定“智能”的搜索工具包。虽然它离能够理解人类的真实想法还差得远，但是它能够在搜索过程中找到意义相近的词，而不仅仅是拼写相近的词。FTS 是 PostgreSQL 原生自带的一个功能模块，不需要单独安装。

FTS 的核心是一个被称为“FTS 配置库”的东西。这个库记录了词与词之间的语义匹配规则，其依据是一本或者多本词典。例如，如果你的词典将 love、romance、infatuation、lust 这几个词当成同义词，那么把其中的一个作为关键词进行搜索时，带有其他几个词的文本也会被认为是符合搜索条件的。词典也会把主干相同的词当成同义词，例如 love、loving、loved 就是主干相同的词。词典还会将同一个动词的不同时态下的分词作为同义词处理，比如 eat、eats、ate、eaten 就会被认为是同义词。

词典中还可以包含停止词，它是指一段话中意义不大的那部分内容。冠词、连接词、介词、代词都是停止词，比如 a、the、on、that 等。

如上所述，FTS 能够实现同义词搜索以及对无意义停止词的过滤，此外它还能够设置搜索结果的排名规则。FTS 可以根据词与词之间的邻近程度以及关键词出现的频度来为搜索结

注 2：英文中“女同性恋、男同性恋、双性人、变性人”合在一起的简称。——译者注

果排名。例如，如果你对描写 romance 和 campus 的电影感兴趣，可以同时搜索 romance 加 campus，另外还可以指定这两个关键词必须作为两个单独的词存在，而且可以指定对于同时包含这两个关键词的搜索结果的排名应更靠前。FTS 还支持为同一关键词出现的位置不同而给予不同的权重评分。例如，如果电影的标题、副标题或者剧情介绍中都可能出现 romance 字样，你可以让标题或者副标题中出现 romance 字样的电影的搜索结果排名更靠前。

5.8.1 FTS配置库

大多数 PostgreSQL 发行版中都自带了 10 个以上的 FTS 配置库，这些配置库都安装在 `pg_catalog` schema 中。

可以通过 `SELECT cfgname FROM pg_ts_config;` 语句或者是 psql 的 `\dF` 命令来查出所有已安装的配置库。一般情况下查询结果如下：

```
cfgname
----------
simple
danish
dutch
english
finnish
french
german
hungarian
italian
norwegian
portuguese
romanian
russian
spanish
swedish
turkish
(16 rows)
```

用户如需创建自定义的配置库或者词典，可以参考 PostgreSQL 官方手册中"全文搜索配置库"和"全文搜索词典"这两部分的内容。

PostgreSQL 并不要求用户使用系统自带的 FTS 配置库，而是允许用户创建自定义的 FTS 配置库。但在真正开始创建自定义配置库之前，建议你查看一下是否已经有别的用户创建了能够满足你要求的配置库，这样你就不用自己重弄一遍了。如果你要做检索的目标文本属于医学领域，你可能会发现已经有人创建好了这样的 FTS 词典，其中记录了大量的医学术语；如果你要检索西班牙文的文本，也可以找到为你所要的西班牙方言量身定做的配置库。

一旦确定了要添加的目标配置库，安装非常简单，一般也不需要进行编译。我们以著名的 hunspell 配置库为例来演示安装过程。

请先从 hunspell_dicts 下载 hunspell 配置库。该库支持多种语言，我们选择安装其中的 hunspell_en_us。

(1) 下载对应目录中的所有文件。
(2) 将下载的 en_us.affix 和 en_us.dict 这两个文件复制到 PostgreSQL 安装目录下的 share/tsearch_data 子目录中。
(3) 将 hunspell_en_us--*.sql 和 hunspell_en_us.control 文件复制到 PostgreSQL 安装目录下的 share/extension 子目录中。

然后执行以下命令：

```
CREATE EXTENSION hunspell_en_us SCHEMA pg_catalog;
```

然后在 psql 中执行示例 5-42，就可以看到刚刚安装的 hunspell 配置库和词典的细节。

示例 5-42 FTS 的 hunspell 配置库

```
\dF+ english_hunspell;

Text search configuration "pg_catalog.english_hunspell"
Parser: "pg_catalog.default"
Token           | Dictionaries
----------------+-------------------------------
asciihword      | english_hunspell,english_stem
asciiword       | english_hunspell,english_stem
email           | simple
file            | simple
float           | simple
host            | simple
hword           | english_hunspell,english_stem
hword_asciipart | english_hunspell,english_stem
hword_numpart   | simple
hword_part      | english_hunspell,english_stem
int             | simple
numhword        | simple
numword         | simple
sfloat          | simple
uint            | simple
url             | simple
url_path        | simple
version         | simple
word            | english_hunspell,english_stem
```



请注意，不是所有 FTS 配置库的安装方法都完全相同，请按照各自的安装手册来安装。

示例 5-43 中的命令用来查看 PostgreSQL 自带的 English 配置库的详情，输出结果中会展示使用了哪些词典。可以将这里的输出内容与前面 hunspell 配置库的输出内容做个对比。

示例 5-43 FTS 的 English 配置库

```
\dF+ english;

Text search configuration "pg_catalog.english"
Parser: "pg_catalog.default"
```

```
Token           | Dictionaries
----------------+--------------
asciihword      | english_stem
asciiword       | english_stem
email           | simple
file            | simple
float           | simple
host            | simple
hword           | english_stem
hword_asciipart | english_stem
hword_numpart   | simple
hword_part      | english_stem
int             | simple
numhword        | simple
numword         | simple
sfloat          | simple
uint            | simple
url             | simple
url_path        | simple
version         | simple
word            | english_stem
```

可以看到，二者唯一的区别就是 hunspell 配置库额外依赖了一个名为 english_hunspell 的词典。

如果想了解哪个配置库是系统当前默认使用的库，可以这样查询：

```
SHOW default_text_search_config;
```

如果希望修改默认配置，可以使用这个命令：

```
ALTER DATABASE postgresql_book
SET default_text_search_config = 'pg_catalog.english';
```

替换后该参数是在 database 级别生效的，但它也可以在服务器级、用户级或者会话级生效，这一点和其他配置参数没什么不同。

5.8.2　TSVector原始文本向量

原始文本必须先被向量化然后才能通过 FTS 对其进行全文检索，向量化以后的内容需存储在一个单独的向量字段中，该向量字段使用的数据类型是 `tsvector`。要从原始文本中生成 `tsvector` 向量字段，需要先指定使用哪个 FTS 配置库。原始文本经过向量化处理以后会变成一个很精简的单词库，这个库中的每个词都被称为“词素”（lexeme，即不能再拆解的单词或词组，如果拆解将失去原来的含义），同时这个库已经剔除了前面介绍过的停止词。`tsvector` 字段中记录了每个词素在原始文本中出现的位置。一个词出现的次数越多，其权重值也就越大。这样每个词素都会对应至少一个位置信息，如果出现多次则对应多个位置信息，看起来就像是一个可变长或变短的向量，这也是 tsvector 这个名字的由来。

可以使用 `to_tsvector` 函数来对一个大文本对象进行向量化。默认情况下，该函数会使用系统默认的 FTS 配置库，当然你也可以显式指定使用另一个 FTS 配置库。

示例 5-44 演示了基于不同的 FTS 配置库将同一段文本向量化以后得到的结果有何差异。

示例 5-44　基于不同的 FTS 配置库对文本进行向量化操作

```
SELECT
    c.name,
    CASE
        WHEN c.name ='default' THEN to_tsvector(f.t)
        ELSE to_tsvector(c.name::regconfig,f.t)
    END As vect
FROM (
    SELECT 'Just dancing in the rain. I like to dance.'::text) As f(t), (
        VALUES ('default'),('english'),('english_hunspell'),('simple')
    ) As c(name);
name             | vect
-----------------+------------------------------------------------------------------
default          | 'danc':2,9 'like':7 'rain':5
english          | 'danc':2,9 'like':7 'rain':5
english_hunspell | 'dance':2,9 'dancing':2 'like':7 'rain':5
simple           | 'dance':9 'dancing':2 'i':6 'in':3 'just':1 'like':7
'rain':5 'the':4 'to':8
(4 rows)
```

从示例 5-44 中可以看出基于四个不同的 FTS 配置库得到的文本向量化结果有何不同。请注意其中的 English 和 Hunspell 配置库剔除了所有像 just 和 to 这样的停止词，此外还按照词典中记录的信息对部分单词进行了规范化处理，比如 dancing 变成了 danc 或者 dance。Simple 配置库不识别词干和停止词，因此其向量化结果中就会出现同一个词的多种时态和那些停止词。

可以看到，`to_vector` 函数的输出结果中还包含每个词素在原始文本中出现的位置。比如，`'danc':2,9` 就表示 dancing 和 dance 这两个词分别出现在原始文本中的第二个词和第九个词的位置。

要想在你的 database 中支持 FTS 能力，需要在存储原始文本的表上增加一个 `tsvector` 类型的向量字段，然后通过定时任务定期更新该向量字段或者在表上创建个触发器，一旦原始文本字段发生了修改，则同时更新向量字段。

我们通过一些虚构的电影数据来进行演示。用 psql 工具可以执行其中的 film.sql 脚本，语法如下：

```
\encoding utf8;
\i film.sql
```

然后我们为 `film` 表添加一个 `tsvector` 向量字段，并生成其内容，如示例 5-45 所示。

示例 5-45　增加 `tsvector` 向量字段并设置词素权重

```
ALTER TABLE film ADD COLUMN fts tsvector;
UPDATE film
SET fts =
    setweight(to_tsvector(COALESCE(title,'')),'A') ||
    setweight(to_tsvector(COALESCE(description,'')),'B');
CREATE INDEX ix_film_fts_gin ON film USING gin (fts);
```

示例 5-45 中，我们对 `title` 和 `description` 字段进行了向量化并把结果存入了新增的向量

字段。为了使查询更快，我们还在向量字段上创建了一个 GIN 类型的索引。GIN 是无损索引，你也可以对其创建一个 GiST 类型的有损索引。相比 GIN 索引，GiST 索引的特点是原始数据信息可能会丢失并且访问速度会慢一些，但它的构造速度很快而且占用磁盘空间也更小。相关内容在 6.3 节中有更详细的介绍。

可以看到，我们在为 fts 字段生成数据的过程中还使用了另外两个前面没介绍过的语法：setweight 函数和连接运算符 ||。

为了区分不同词素的重要程度，我们引入了权重（weight）的概念，每个词素都有自己的权重。权重值只能是 A、B、C 和 D 四类中的一种，A 类表示重要程度最高，在搜索结果中也需要排名最靠前。在示例 5-45 中，我们为从原始文本的 title 字段中抽取出来的词素赋予了 A 类权重，然后为 description 字段中的词素赋予了 B 类权重。如果我们搜索的关键词命中了 title 中的某个词素，那么我们认为这个搜索结果比从 description 字段抽取出来的词素命中的结果相关性更高，更符合用户的需要。

多个 tsvector 向量可以通过连接运算符 || 合并为一个新的 tsvector。我们在前面的例子中使用了该运算符将 title 和 description 中生成的向量合并为一个，这样真正执行全文检索时，只需要搜索一个向量字段即可。

如果原始文本字段的内容发生了改变，那么必须对其重新进行向量化。为了避免每次都要手动执行 to_tsvector 来重新向量化，可以针对更新操作建立一个触发器。这个触发器可以基于下面这个非常好用的触发器函数来构造，如示例 5-46 所示。

示例 5-46　能够自动更新向量字段的触发器

```
CREATE TRIGGER trig_tsv_film_iu
BEFORE INSERT OR UPDATE OF title, description ON film FOR EACH ROW
EXECUTE PROCEDURE tsvector_update_trigger(fts,'pg_catalog.english',
title,description);
```

一旦在 title 或 description 字段上执行了 insert 或者 update 操作，则该触发器会被触发，然后会自动完成重新向量化的工作。这个方法有个缺点，就是里面调用的 tsvector_update_trigger 函数不支持设置权重。

5.8.3　TSQueries检索条件向量

对于 FTS 或者任何别的文本检索方法来说，都必须具备两个基本元素：一个是被检索的原始文本，一个是检索条件（或者说关键词）。对 FTS 来说，原始文本和检索条件都必须先被向量化然后才能使用。前面已经介绍过了如何针对原始文本创建 tsvector 向量字段，接下来将介绍如何对检索条件进行向量化处理。

在 FTS 检索机制中，用 tsquery 类型来表示向量化以后的检索条件。PostgreSQL 提供了若干函数来实现检索条件的向量化处理，包括 to_tsquery、plainto_tsquery 和 phraseto_tsquery。其中 phraseto_tsquery 是 PostgreSQL 9.6 新引入的，相比其他两个，该函数会将检索条件中各关键词的顺序也考虑在内。

检索条件向量一般是运行时临时生成的，不会放到表中存储下来。然而，如果你的系统支持用户把自定义的检索条件存储下来供下次重用，那么你可以把解析后的向量数据用

tsquery 类型的字段存下来。

示例 5-47 演示了 to_tsquery 函数的输出结果，分别基于两个配置库：默认的 English 库和我们后加的 Hunspell 库。

示例 5-47　用 to_tsquery 函数来构造 tsquery 向量

```
SELECT to_tsquery('business & analytics');

to_tsquery
----------------
'busi' & 'analyt'

SELECT to_tsquery('english_hunspell','business & analytics');

to_tsquery
-------------------------------
('business' | 'busy') & 'analyt'
```

这两个例子都是用 business 和 analytics 两个词来进行全文检索，其中的与运算符（&）表示两个关键词必须在目标文本中同时出现才算检索命中。或运算符（|）表示目标文本中必须出现两个词中的至少一个就算检索命中。如果使用的配置库查到其中的某个关键词拥有多个词干，那么最终的向量化结果中会用 | 运算符把这些词干连接到一起呈现。



在对检索条件进行向量化处理时，使用的配置库应该与对原始文本向量化时使用的配置库相同。

plainto_tsquery 是 to_tsquery 的变种，它会自动在检索条件的关键词之间加上 & 运算符，可以为你节省一点时间。语法如示例 5-48 所示。

示例 5-48　用 plainto_tsquery 函数来构造 tsquery 向量

```
SELECT plainto_tsquery('business analytics');

plainto_tsquery
----------------
'busi' & 'analyt'
```

to_tsquery 和 plainto_tsquery 仅考虑关键词本身，不考虑其先后位置。因此对这两个函数来说，“business analytics”和“analytics business”得到的 tsquery 结果是一样的。由于无法识别词与词之间的先后顺序，带来的问题就是用户只能搜索单个关键词。PostgreSQL 9.6 中为了解决这个问题而引入了 phraseto_tsquery 函数。在示例 5-49 中可以看到，phraseto_tsquery 函数的向量化结果中在词素与词素之间增加了距离运算符，这意味着在检索目标文本中 business 和 analytics 两个词必须以一定顺序出现时才算命中。这样就从功能上实现了从单词搜索到词组搜索的进化。

示例 5-49　用 phraseto_tsquery 函数来构造 tsquery 向量

```
SELECT phraseto_tsquery('business analytics');

phraseto_tsquery
```

```
-------------------
'busi' <-> 'analyt'

SELECT phraseto_tsquery('english_hunspell','business analytics');

phraseto_tsquery
---------------------------------------------
'business' <-> 'analyt' | 'busy' <-> 'analyt'
```

你还可以直接使用类型强转语法将文本串转为 tsquery 数据，这样就无须使用前面介绍的几个函数，语法类似 'business & analytics'::tsquery。这样做虽然简单，但也意味着无法利用前述函数提供的功能，检索关键词不会被抽取为词素，只会被简单地原文复制。

多个 tsquery 向量数据可以使用或运算符（||）或者与运算符（&&）连接在一起。tsquery1 || tsquery2 表达式的意思是检索目标文本要么满足 tsquery1 条件，要么符合 tsquery2 条件。tsquery1 && tsquery2 表达式的意思是检索目标文本必须同时满足 tsquery1 和 tsquery2 条件。

示例 5-50 分别演示了这两种情况。

示例 5-50　多个 tsquery 向量的关联

```
SELECT plainto_tsquery('business analyst') || phraseto_tsquery('data scientist');

tsquery
-------------------------------------------
'busi' & 'analyst' | 'data' <-> 'scientist'

SELECT plainto_tsquery('business analyst') && phraseto_tsquery('data scientist');

tsquery
--------------------------------------------
'busi' & 'analyst' & ('data' <-> 'scientist')
```

tsquery 和 tsvector 这两种数据类型还支持其他一些运算符，比如判定一个向量值是否是另一个向量值的子集等。详情请参考官方手册中“全文检索函数和运算符”一节的内容。

5.8.4　使用全文检索

前面已经为原始文本创建好了 tsvector 向量数据，也为检索条件创建好了 tsquery 向量数据，现在可以真正地执行全文检索了。FTS 检索使用的是 @@ 运算符，如示例 5-51 所示。

示例 5-51　进行 FTS 检索

```
SELECT left(title,50) As title, left(description,50) as description
FROM film
WHERE fts @@ to_tsquery('hunter & (scientist | chef)') AND title > '';
title                   | description
------------------------+---------------------------------------------------
ALASKA PHANTOM          | A Fanciful Saga of a Hunter And a Pastry Chef who
CAUSE DATE              | A Taut Tale of a Explorer And a Pastry Chef who mu
CINCINATTI WHISPERER    | A Brilliant Saga of a Pastry Chef And a Hunter who
COMMANDMENTS EXPRESS    | A Fanciful Saga of a Student And a Mad Scientist w
DAUGHTER MADIGAN        | A Beautiful Tale of a Hunter And a Mad Scientist w
GOLDFINGER SENSIBILITY  | A Insightful Drama of a Mad Scientist And a Hunter
```

```
HATE HANDICAP          | A Intrepid Reflection of a Mad Scientist And a Pio
INSIDER ARIZONA        | A Astounding Saga of a Mad Scientist And a Hunter
WORDS HUNTER           | A Action-Packed Reflection of a Composer And a Mad
(9 rows)
```

示例 5-51 中查出了所有 `title` 和 `description` 字段中含有 hunter 这个词，并且还含有 scientist 或者 chef 这两个单词中的一个或者两个的所有电影。

如果你当前使用的是 PostgreSQL 9.6，那么还可以指定检索条件内部的关键词之间的接近度和出现的先后顺序，如示例 5-52 所示。

示例 5-52　指定了关键词先后顺序和接近度的 FTS 检索

```
SELECT left(title,50) As title, left(description,50) as description
FROM film
WHERE fts @@ to_tsquery('hunter <4> (scientist | chef)') AND title > '';

title            | description
-----------------+---------------------------------------------------
ALASKA PHANTOM   | A Fanciful Saga of a Hunter And a Pastry Chef who
DAUGHTER MADIGAN | A Beautiful Tale of a Hunter And a Mad Scientist w
(2 rows)
```

示例 5-52 中要求 hunter 在 scientist 或者 chef 之前，而且它们之间必须相差 4 个词。

5.8.5　对检索结果进行排序

FTS 支持对检索结果进行排序，具体是通过 `ts_rank` 和 `ts_rank_cd` 这两个函数来实现的。`ts_rank` 函数只考虑检索关键词在目标文本中出现的频率和权重，而 `ts_rank_cd`（cd 代表 coverage density，即覆盖密度）还考虑了关键词在目标文本中出现的位置。检索条件中的词素在被检索文本中出现的位置越靠近，则该条检索结果的相关度越高，最终排名越靠前。只有当原始文本的 `tsvector` 向量数据中带有位置标记时，使用 `ts_rank_cd` 函数才有意义，因为没有位置标记的情况下根本无法计算检索关键词之间的距离，此时该函数的返回值是 0。另外，我们很容易就能想到检索关键词出现的频率也需要根据位置标记才能计算出来。因此，如果被检索的原始文本的 `tsvector` 向量数据中没有提供位置标记的话，`ts_rank` 函数就只能依赖权重来排名了。默认情况下，`ts_rank` 和 `ts_rank_cd` 函数在计算过程中会将权重值 A、B、C、D 映射为 1.0、0.4、0.2 和 0.1，之后再进行计算。示例 5-53 使用了默认的排序参数。

示例 5-53　对检索结果进行排序

```
SELECT title, left(description,50) As description,
    ts_rank(fts,ts)::numeric(10,3) AS r
FROM film, to_tsquery('english','love  & (wait | indian | mad)') AS ts
WHERE fts @@ ts AND title > ''
ORDER BY r DESC;
title         | description                                        | r
--------------+----------------------------------------------------+------
INDIAN LOVE   | A Insightful Saga of a Mad Scientist And a Mad Sci | 0.999
LAWRENCE LOVE | A Fanciful Yarn of a Database Administrator And a  | 0.252
(2 rows)
```

假设我们希望只有当检索条件字段出现在 title 字段中时才算命中，那么我们可以将 title 的权重值设置为 1，其他字段的权重值都设为 0。接下来用示例 5-54 再实现一遍示例 5-53 相同的检索逻辑，唯一的差别是这次传入了一组权重值。

示例 5-54　使用用户自定义的权重值来对检索结果进行排序

```
SELECT
    left(title,40) As title,
    ts_rank('{0,0,0,1}'::numeric[],fts,ts)::numeric(10,3) AS r,
    ts_rank_cd('{0,0,0,1}'::numeric[],fts,ts)::numeric(10,3) As rcd
FROM film, to_tsquery('english', 'love & (wait | indian | mad )') AS ts
WHERE fts @@ ts AND title > ''
ORDER BY r DESC;

title         | r     | rcd
--------------+-------+------
INDIAN LOVE   | 0.991 | 1.000
LAWRENCE LOVE | 0.000 | 0.000
(2 rows)
```

请注意：上面输出的第二行结果中的 rank 值都是 0，这是因为该记录的 title 字段内容并不完全满足 tsquery 条件的要求。



如果你对检索性能非常在意，那么我们建议在查询语句中显式地指明 FTS 配置库，而不要依赖默认值。根据 Oleg Bartunov 的博文“Some FTS Tricks”中的介绍，使用 to_tsquery('english','social & (science | scientist)') 这种写法会比使用 to_tsquery('social & (science| scientist)') 这种写法快一倍。

5.8.6　全文检索向量信息的裁减

默认情况下，对原始文本进行向量化处理时会自动加上位置标记（即词素在原始文本中出现的位置）以及可选的权重值（A、B、C、D 四档）信息。但如果你的检索目标是只要匹配了检索条件中的关键词就可以，而完全不关心这些结果中关键词出现的位置、出现的频率以及重要性，那么你完全可以用 strip 函数对 tsvector 向量数据进行裁减，这样可以节省磁盘空间，也可以提升查询速度。示例 5-55 比较了裁减前和裁减后向量数据的差异。

示例 5-55　裁减前和裁减后向量数据的比较

```
SELECT fts
FROM film
WHERE film_id = 1;

'academi':1A 'battl':15B 'canadian':20B 'dinosaur':2A 'drama':5B 'epic':4B
'feminist':8B 'mad':11B 'must':14B 'rocki':21B 'scientist':12B 'teacher':17B

SELECT strip(fts)
FROM film
WHERE film_id = 1;

'academi' 'battl' 'canadian' 'dinosaur' 'drama' 'epic' 'feminist' 'mad'
'must' 'rocki' 'scientist' 'teacher'
```

请务必牢记：虽然裁减后的 tsvector 向量数据查询起来更快，占用磁盘空间更少，但很多运算符和函数都不支持与这种裁减后的向量数据配合使用。例如，由于裁减后的向量数据中不含位置标记信息，因此就不可以对其使用距离运算符。

5.8.7　全文检索机制对JSON和JSONB数据类型的支持

PostgreSQL 10 中为 ts_headline 和 to_tsvector 函数新增了可以处理 json 和 jsonb 数据的版本。这两个函数的 json/jsonb 入参版本的使用方式与其 text 入参版本的用法完全相同，唯一值得注意的地方是它们仅处理 json/jsonb 数据中的 value 部分，而不会关注 key 部分以及 json 本身的那些标记符。示例 5-56 用这两个函数对示例 5-28 中创建的表中的 person 字段进行了处理。

示例 5-56　对 json/jsonb 数据进行 tsvector 向量化处理

```
SELECT to_tsvector(person)
   FROM persons WHERE id=1;

      to_tsvector
---------------------------------------------------------------------------------
 '-5083':19 '-6719':13 '-722':12 '-852':18 '619':11,17 'alex':3 'azaleah':25
 'brandon':21 'cell':15 'm':23 'ofelia':7 'rafael':5 'sonia':1 'work':9
(1 row)
```

上例演示的是处理 json 类型，如果要演示处理 jsonb 类型，只需把脚本中的 person 换成 person_b 即可。json、jsonb 入参版本的 ts_headline 和 to_tsvector 函数各自还有一个变体，其第一个参数可用于指定用哪个 FTS 配置库，这个特点与 text 入参版本的 ts_headline 和 to_tsvector 函数完全相同。为了更好地利用这些函数的能力，我们推荐的做法是在被检索的 json/jsonb 数据所在的表上先增加 tsvector 向量字段，然后创建触发器在修改时自动生成或者在需要时手工生成向量数据。

如前所述，ts_headline 函数也有了支持 json/jsonb 入参的版本，该函数的功能是将 json/jsonb 数据中所有命中的文本都标记为 HTML 格式。示例 5-57 对 JSON 文档数据中所有的 Rafael 文本都做了标记。

示例 5-57　对检索命中的文本打上标记

```
SELECT ts_headline(person->'spouse'->'parents', 'rafael'::tsquery)
FROM persons_b WHERE id=1;

{"father": "<b>Rafael</b>", "mother": "Ofelia"}
(1 row)
```

请注意，上面输出的命中结果前后已打上了 HTML 的 <b> 标签。

5.9　自定义数据类型和复合数据类型

本节将介绍如何定义和使用自定义数据类型。composite（也称为 record、row）常用于构建需要转为自定义数据类型的对象或者是作为需要返回多个字段的函数的返回值类型定义。

5.9.1 所有表都有一个对应的自定义数据类型

PostgreSQL 在建表时会自动创建一个与表结构完全相同的自定义数据类型，而且这种类型与其他的原生数据类型在使用上毫无区别。可以在建表时指定某字段为表类型或者表数组类型，也就是说可以把一张表的字段定义为另一张表。这种将表层层嵌套的用法与“turducken”（特大啃）[3] 很像。在示例 5-58 中，我们将用“特大啃”的例子来演示。

示例 5-58 为“特大啃”建嵌套表

```
CREATE TABLE chickens (id integer PRIMARY KEY);
CREATE TABLE ducks (id integer PRIMARY KEY, chickens chickens[]);
CREATE TABLE turkeys (id integer PRIMARY KEY, ducks ducks[]);

INSERT INTO ducks VALUES (1, ARRAY[ROW(1)::chickens, ROW(1)::chickens]);
INSERT INTO turkeys VALUES (1, array(SELECT d FROM ducks d));
```

上面我们直接在 ducks 表的一条记录的 chickens 字段中插入了两条 chickens 记录，这种情况下这两条记录的构造不受 chickens 表定义的约束，因此即使它们的主键重复也没关系。我们生成了两条 chickens 记录，填入 ducks 表中，然后将这条 ducks 记录填入到 turkeys 表中，这个过程相当于把两只 chicken 塞入一只 duck，然后再把这只 duck 塞入一只 turkey，跟制作“特大啃”的过程是完全一样的。

最后看一下得到的 turkeys 记录是什么样子的：

```
SELECT * FROM turkeys;

output
------------------------
id | ducks
---+----------------------
1  | {"(1,\"{(1),(1)}\")"}
```

嵌套表中内嵌的表记录是可以进行修改的。例如，我们要对第一个 turkey 内嵌的第二个 chicken 进行修改，那么可以执行如下操作：

```
UPDATE turkeys SET ducks[1].chickens[2] = ROW(3)::chickens
WHERE id = 1 RETURNING *;

output
--------------
id | ducks
---+----------------------
 1 | {"(1,\"{(1),(3)}\")"}
```

我们使用 RETURNING 子句来返回本次更新操作涉及的所有记录，我们将在 7.2.10 节中介绍 RETURNING 子句的用法。

一个复合类型的记录行或者字段不管其内部结构有多么复杂，都可以被转换为一个 json 或者 jsonb 类型的字段，语法如下：

注 3：turducken 是 turkey-duck-chicken 的简称，是一种美食新吃法：将无骨鸡填到无骨鸭的肚子里，然后再将填了无骨鸡的无骨鸭填到无骨火鸡的肚子里。这种食物很好地体现了多层嵌套的关系。——译者注

```
SELECT id, to_jsonb(ducks) AS ducks_jsonb
FROM turkeys;

id | ducks_jsonb
---+---------------------------------------------
 1 | [{"id": 1, "chickens": [{"id": 1}, {"id": 3}]}]
(1 row)
```

PostgreSQL 内部维护着数据库对象之间的依赖关系。前述 ducks 表的 chickens 字段依赖于 chickens 表，turkeys 表的 ducks 记录依赖于 ducks 表。要想删除 chickens 表有两种方法，要么在 drop 语句中带上 CASCADE 关键字，要么先删除 ducks 表中的 chickens 字段。如果使用前一种方法，那么 ducks 表的 chickens 字段会被自动删除，而且此过程中无告警信息。相应地，turkeys 表的 ducks 字段的定义也将自动跟着改变。

5.9.2 构建自定义数据类型

尽管仅仅通过创建表就可以轻松创建复合数据类型，但有时候我们仍需要从头开始构建自己的数据类型。例如，使用以下语句可以构建一个复数数据类型：

```
CREATE TYPE complex_number AS (r double precision, i double precision);
```

可以将此类型作为字段类型定义使用：

```
CREATE TABLE circuits (circuit_id serial PRIMARY KEY, ac_volt complex_number);
```

可以使用如下语法对这个表进行查询：

```
SELECT circuit_id, (ac_volt).* FROM circuits;
```

这种语法也可以：

```
SELECT circuit_id, (ac_volt).r, (ac_volt).i FROM circuits;
```



你可能会问：上面语句中 ac_volt 外面为什么要加括号？如果不加的话，PostgreSQL 会报错说 FROM 子句中找不到 ac_volt 这张表。根据这一点就很好理解了，加括号是为了不让 PostgreSQL 将其理解为表名。

5.9.3 复合类型中的空值处理

在 ANSI SQL 标准中，两个 NULL 值之间不允许进行“值相等”（即 NULL=NULL）或者“值不等”（即 NULL != NULL）判定，这一点经常会给用户带来理解上的困难。当处理 NULL 值时，需要使用 IS NULL、IS NOT NULL 或者 NOT（somevalue IS NULL）这种表达方式。对于基础数据类型来说，something IS NULL 就是 something IS NOT NULL 的反义表达式，但对于复合数据类型来说却并不是这样。

PostgreSQL 在处理 NULL 值时严格遵循 ANSI SQL 标准，其中规定：复合数据类型值的 IS NULL 判定要想成立，其前提是该复合数据类型值的每一个元素都是 NULL，这很合理。但接下来就有点“不太合理”了，复合数据类型值的 IS NOT NULL 判定要想成立，前提是该复合数据类型值的每一个元素都是 NULL，而不是说只需其中任何一个元素不为 NULL 即可。这里特别容易出错，因此请牢记此规则。

5.9.4 为自定义数据类型构建运算符和函数

在构建自定义数据类型后，你自然就需要为其创建相应的函数和运算符。接下来将演示如何为 `complex_number` 类型创建一个 + 运算符，而创建处理函数的方法将在第 8 章中介绍。前面已经介绍过，每个运算符都有一个底层实现函数，该函数需要一个或者两个参数，运算符就是这个函数的符号化别名。在 PostgreSQL 官方手册的"创建运算符"一节中，你可以看到系统允许使用哪些字符来定义新的运算符。

运算符不仅仅是其底层实现函数的别名，它还可以提供一些可以帮助规划器更好工作的优化信息，规划器借助这些信息可以判定如何使用索引，如何以最低的成本访问数据，以及哪些运算符表达式是等价的。这些信息的完整列表以及每一类信息的具体作用，可以参考官方手册中"运算符的优化信息"一节的内容。

创建运算符的第一步是创建其底层实现函数，如示例 5-59 所示。

示例 5-59　为 `complex_number` 创建底层实现函数

```
CREATE OR REPLACE FUNCTION add(complex_number, complex_number)
RETURNS complex_number  AS
$$
    SELECT
        ((COALESCE(($1).r,0) + COALESCE(($2).r,0)),
        (COALESCE(($1).i,0) + COALESCE(($2).i,0)))::complex_number;
$$
language sql;
```

接下来要创建一个运算符来代表此函数，如示例 5-60 所示。

示例 5-60　为 `complex_number` 类型定义 + 运算符

```
CREATE OPERATOR + (
    PROCEDURE = add,
    LEFTARG = complex_number,
    RIGHTARG = complex_number,
    COMMUTATOR = +
);
```

然后我们测试一下这个新的 + 运算符：

```
SELECT (1,2)::complex_number + (3,-10)::complex_number;
```

输出结果是 `(4,-8)`。

虽然我们在此处没有举例说明，但你可以对函数和运算符进行重载，以使其可以接受多种不同类型的输入。例如，你可以创建一个支持 `complex_number` 和 `integer` 相加的 `add` 函数和相应的 + 计算符，这就实现了对原逻辑的扩展。

支持自定义数据类型和运算符让 PostgreSQL 从机制上具有了自我演进的能力，开源社区无数开发人员利用此能力为 PostgreSQL 平台添砖加瓦。随着这个开发平台的羽翼日渐丰满，我们离"一切皆以表驱动"的理想境界也越来越近。

第 6 章

表、约束和索引

表是关系型数据库存储体系的基本单元。设计好结构化的表并且定义表与表之间的关联关系是关系型数据库的核心设计思想。在 PostgreSQL 中，约束定义了表与表之间的关系。与以堆结构存储的记录相比，表的优势就在于有索引。你会在很多书的末尾看到词汇索引表，也会在写字楼的大堂看到每层楼的租户名单，表索引的作用与它们类似，即在表中快速查找到目标数据的位置，这样就不用每次查询时都扫描整张表的内容。

本章将介绍创建表和插入记录的语法。然后将介绍约束的用法，约束可以保证数据库的记录不会违反我们制定的规则。最后将展示如何为表创建索引来加速查询。在表上创建索引是需要经过深思熟虑的，因为一个错误的索引会导致查询效果比全表扫描还差，也就是说创建了还不如不创建。并不是所有的索引都是“生来平等”的，数据库领域的算法专家为不同的数据类型设计出了不同类型的索引，目的是将查询的速度提升到极致。

6.1 表

除了普通的表以外，PostgreSQL 还提供了许多不常见的表，具体包括临时表、无日志表、继承表、基于复合类型的表以及外部表（第 10 章将介绍外部表）。

6.1.1 基本的建表操作

示例 6-1 演示了建表语法，在所有支持 SQL 的数据库中建表语法都是类似的。

示例 6-1 基本的建表操作

```
CREATE TABLE logs (
log_id serial PRIMARY KEY, ❶
user_name varchar(50), ❷
description text, ❸
```

```
log_ts timestamp with time zone NOT NULL DEFAULT current_timestamp
); ❹
CREATE INDEX idx_logs_log_ts ON logs USING btree (log_ts);
```

❶ `serial` 数据类型是一种自增长的数字类型。建表时如果有一个 `serial` 类型的字段，那么系统会自动在 schema 中同时创建一个对应的序列号生成器。`serial` 类型字段的值是一个整型数字，它会自动被赋值为序列号生成器的下一个值。每张表一般来说只会有一个 `serial` 字段，且一般用作主键。对于特别大的表，应该使用 `bigserial` 类型，因为它能容纳的数值上限更大。

❷ `varchar` 是 character varying（可变长字符串）的简写，其定义与其他数据库产品中的定义类似。你可以不为 `varchar` 字段设定最大长度值，此时它与 `text` 类型几乎是一样的。

❸ `text` 是一种不定长度的字符串，无最大长度限制。

❹ `timestamp with time zone`（可简写为 `timestamptz`）是一种表示日期和时间的类型，总是以国际标准时间（UTC）格式存储。该类型在显示时总是以服务器当前所在时区为基准进行显示，当然你也可以要求使用指定的时区进行显示。更多关于此类型的讨论，请参考 5.3.1 节。

PostgreSQL 10 中新增了对 `IDENTITY` 关键字的支持。`IDENTITY` 也可以将字段定义为自增序列号类型，这是一种更加符合标准的语法。

你可以将现有的 `log_id` 字段的 `serial` 类型修改为 `IDENTITY` 类型。与 `serial` 不同的是，`IDENTITY` 类型的底层不再借助一个自动生成的序列号生成器。语法如下：

```
DROP SEQUENCE logs_log_id_seq CASCADE;
ALTER TABLE logs
    ALTER COLUMN log_id ADD GENERATED BY DEFAULT AS IDENTITY;
```

如果此表中已有数据，需要防止序列号再次从 1 生成从而造成重复，语法如下：

```
ALTER TABLE logs
    ALTER COLUMN log_id RESTART WITH 2000;
```

如果是创建新表，要按照如示例 6-2 所示的 `IDENTITY` 语法创建表，不用原来的 `serial` 语法。

示例 6-2　使用 `IDENTITY` 语法创建表

```
CREATE TABLE logs (
log_id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
user_name varchar(50),
description text,
log_ts timestamp with time zone NOT NULL DEFAULT current_timestamp
);
```

示例 6-2 的语法结构与示例 6-1 基本相同，只是更详细。

那么在什么情况下应该使用 `IDENTITY` 替代 `serial` 呢？ `IDENTITY` 语法的主要优点在于一个 identity 总是与所属表绑定的，其值的递增或者重置都是与表本身一体化管理的，不会受其他对象干扰。`serial` 类型则不是这样，它会在后台自动创建一个序列号生成器，这个序列号生成器可以与别的表共享也可以本表独享，当不需要该序列号生成器时需要手动

删除它。如果你需要重置一个 `serial` 类型字段的初始值，需要修改后台那个自动生成的序列号生成器，这也意味着得先知道那个序列号生成器的名字。很显然，这个管理过程比 `IDENTITY` 要繁琐很多。

当需要在多张表之间共享一个递增序列号时，`serial` 类型依然是很有用的。这种情况下，需要创建一个独立的序列号生成器，并把需要共享该序列的每张表的相应字段的默认值设为该序列号生成器的下一个值。从内部实现机制来看，`IDENTITY` 语法与 `serial` 类型的做法其实是类似的，都是自动创建一个序列号生成器，只不过 `IDENTITY` 不会将这个序列号生成器对象暴露给外界去修改。

6.1.2 继承表

PostgreSQL 是唯一提供表继承功能的数据库。如果创建一张表（子表）时指定为继承自另一张表（父表），则建好的子表除了含有自己的字段外还会含有父表的所有字段。PostgreSQL 会记录下这个继承关系，这样一旦父表的结构发生了变化，子表的结构也会自动跟着变化。这种父子继承结构的表可以完美地适用于需要数据分区的场景。当查询父表时，PostgreSQL 会自动把子表的记录也取出来。值得注意的是，并不是所有父表的特征都会被子表继承下来，比如主表的主键约束、唯一性约束以及索引就不会被继承。check 约束会被继承，但子表还可以另建自己的 check 约束（详见示例 6-3）。

示例 6-3　创建继承表

```
CREATE TABLE logs_2011 (PRIMARY KEY (log_id)) INHERITS (logs);
CREATE INDEX idx_logs_2011_log_ts ON logs_2011 USING btree(log_ts);
ALTER TABLE logs_2011
    ADD CONSTRAINT chk_y2011
CHECK (
    log_ts >= '2011-1-1'::timestamptz AND log_ts < '2012-1-1'::timestamptz
); ❶
```

❶ 我们定义了一个 check 约束来限制只能录入 2011 年的数据。该 check 约束告诉查询规划器在查询父表时跳过不满足条件的子表。

PostgreSQL 9.5 新增支持了在本地表和外部表之间做表继承，而且可以互相继承。支持该特性主要是为了实现表的分布式存储。

6.1.3 原生分区表支持

PostgreSQL 10 中增加了对原生分区表功能的支持。原生分区表从使用上看与之前的表继承功能非常类似，都允许把数据切分到多张独立的子表中，并且规划器会选择性地跳过不符合查询条件的子表。从内部实现机制看，二者是非常像的，只不过使用的 DDL 语法不同。

尽管在很多场景下分区表都可以替代原来的表继承功能，但还不能完全替代。以下是表继承和分区表这两个功能的关键区别。

- 分区表使用声明式的 `CREATE TABLE .. PARTITION BY RANGE ..` 语法，后台会默认创建一个分区表组。

- 使用分区表时，如果对主表插入数据，记录会按照分区规则被路由到相应的分区中。但使用表继承时情况就不是这样，你需要直接把数据插入正确的子表中，或者在主表上挂载触发器来把记录路由到子表中。
- 分区表的所有子分区都必须具备相同的字段结构，而表继承中的子表完全可以比父表拥有更多字段。
- 分区表的每个分区都隶属于一个共同的分区表组，这意味着它们只能有一个父表，然而表继承功能中的一张子表可以继承自多个父表。
- 分区表的父表是一个逻辑对象而非物理实体，因此它上面不可以定义主键、唯一键或者索引，但是每个子分区可以定义这些。表继承机制中的情况与此不同：父表和每个子表都可以有主键，而且主键只需在本表内部唯一即可，并不一定要在所有子表范围内都唯一。
- 与表继承机制中的父表不同，分区表的父表不能存储自己的记录。所有针对父表插入的记录都会被路由到相应的子分区中，如果没有符合条件的子分区则会报错。

我们将使用分区表语法重建示例 6-1 中的 `logs` 表，其中会用分区语法来创建子分区，而不会使用示例 6-3 所示的表继承语法。

首先，删除现有的 `logs` 表及其所有子表：

```
DROP TABLE IF EXISTS logs CASCADE;
```

创建分区表时，必须使用 `PARTITION BY` 语法来表明这是一个分区表，语法如示例 6-4 所示。可以对比看一下示例 6-1，其中我们先创建了一张普通表。另外，请注意我们并没有定义主键，因为分区表的主表并不支持主键。

示例 6-4　创建分区表的主表

```
CREATE TABLE logs (
log_id int GENERATED BY DEFAULT AS IDENTITY,
user_name varchar(50),
description text,
log_ts timestamp with time zone NOT NULL DEFAULT current_timestamp
) PARTITION BY RANGE (log_ts);
```

与表继承机制类似的是，分区表机制中也需要单独创建子分区表，不同之处是使用了 `FOR VALUES` 语法来指明每张子表能容纳的数据范围，而表继承中使用的是 check 约束。我们在示例 6-5 中把示例 6-3 的内容重演一遍，只不过采用的是 `FOR VALUES FROM` 语法而非 `INHERITS` 语法。

示例 6-5　创建子分区

```
CREATE TABLE logs_2011 PARTITION OF logs ❶
FOR VALUES FROM ('2011-1-1') TO ('2012-1-1') ❷;
CREATE INDEX idx_logs_2011_log_ts ON logs_2011 USING btree(log_ts); ❸
ALTER TABLE logs_2011 ADD CONSTRAINT pk_logs_2011  PRIMARY KEY (log_id) ❹;
```

❶ 定义一张新表，作为 `logs` 表的一个子分区。

❷ 定义该分区中能够存储的数据范围。子分区之间的数据存储范围不能有重叠，如果违反此规则，子分区的 `CREATE TABLE` 语句会失败。

❸❹ 子分区表上可以定义索引和主键。与表继承类似的是，子分区表的主键并不需要在所有子分区表中全局唯一。

如果尝试插入如下的记录：

```
INSERT INTO logs(user_name, description ) VALUES ('regina', 'Sleeping');
```

语句会因找不到合适的分区而报这个错：

```
ERROR:  no partition of relation "logs" found for row
DETAIL:  Partition key of the failing row contains
(log_ts) = (2017-05-25 02:58:28.057101-04).
```

接下来为当前年份再创建一个分区：

```
CREATE TABLE logs_gt_2011 PARTITION OF logs
FOR VALUES FROM ('2012-1-1') TO (unbounded);
```

这里与示例 6-5 不同的地方是使用了 unbounded 关键字来表示分区的截止范围，这样将来的日期也都能匹配到这个分区。

再执行一次前面的插入语句，这次会成功。执行 SELECT * FROM logs_gt_2011; 查询就可以看到记录被路由到了新建的分区中。

请注意，在实际项目中不是只创建完分区就可以，我们需要针对新建的分区创建索引和主键以提升查询效率。

与表继承机制类似的是，当查询父表时，所有不符合条件的子分区都会被跳过，如示例 6-6 所示。

示例 6-6　规划器自动跳过不符合条件的分区

```
EXPLAIN ANALYZE SELECT * FROM logs WHERE log_ts > '2017-05-01';

Append  (cost=0.00..15.25 rows=140 width=162)
(actual time=0.008..0.009 rows=1 loops=1)
  ->  Seq Scan on logs_gt_2011  (cost=0.00..15.25 rows=140 width=162)
(actual time=0.008..0.008 rows=1 loops=1)
        Filter: (log_ts > '2017-05-01 00:00:00-04'::timestamp with time zone)
Planning time: 0.152 ms
Execution time: 0.022 ms
```

如果你使用的是 PostgreSQL 10 自带的 PSQL，针对分区表执行表结构查询命令时将得到更为详细的信息，在这些信息中能看到每个分区能容纳数据的范围。

```
\d+ logs

Table "public.logs"
:
Partition key: RANGE (log_ts)
Partitions: logs_2011
    FOR VALUES FROM ('2011-01-01 00:00:00-05') TO ('2012-01-01 00:00:00-05'),
            logs_gt_2011
            FOR VALUES FROM ('2012-01-01 00:00:00-05') TO (UNBOUNDED)
```

6.1.4　无日志表

对于发生磁盘故障或者系统崩溃后可以被重建的临时数据来说，其操作速度比可靠性更重要。PostgreSQL 从 9.1 版开始支持 UNLOGGED 修饰符，使用该修饰符可以创建无日志的表，

如示例 6-7 所示。系统不会为这种表记录任何事务日志（业界一般也称为 WAL 日志，即 write-ahead log）。无日志表的一大优势是其写入记录的速度远远超过普通的有日志表，依照我们的经验，大概会快 10 到 15 倍。

如果你的服务器不小心被掉电重启，那么无日志表中的数据会在事务回滚过程中被全部清除。无日志表的另一个特性是它无法被纳入 PostgreSQL 的复制机制，因为复制机制依赖事务日志。`pg_dump` 有一个选项可以允许你跳过备份无日志的表。

示例 6-7　创建无日志表

```
CREATE UNLOGGED TABLE web_sessions (
        session_id text PRIMARY KEY,
        add_ts timestamptz,
        upd_ts timestamptz,
        session_state xml);
```

使用无日志表还有一些别的缺点。在 PostgreSQL 9.3 之前，无日志表不支持 GiST 索引（参考 6.3.1 节），该类型的索引一般适用于高级的数据类型，比如数组、范围、JSON、全文检索以及空间类型等。不过任何 PostgreSQL 版本中的无日志表都可以使用 B-树索引和 GIN 索引。

在 PostgreSQL 9.5 之前，要想把无日志表改为有日志表是很麻烦的。9.5 版之后，只需执行以下命令即可：

```
ALTER TABLE some_table SET LOGGED;
```

6.1.5　TYPE OF

PostgreSQL 在创建一张表时，会自动在后台创建一个结构完全相同的复合数据类型，反之则不是这样。你可以使用一个复合数据类型来作为建表的模板。下面将演示该功能，首先创建一个复合数据类型。

```
CREATE TYPE basic_user AS (user_name varchar(50), pwd varchar(10));
```

然后可以创建一张表，表结构就是该复合类型，如示例 6-8 所示。

示例 6-8　以复合数据类型为模板来创建一张表

```
CREATE TABLE super_users OF basic_user (CONSTRAINT pk_su PRIMARY KEY (user_name));
```

当基于数据类型来创建表时，你不能指定表字段的定义，一切以数据类型本身的定义为准。然而，为复合数据类型新增或者移除字段时，PostgreSQL 会自动修改相应的表结构。这种机制的优点是，如果你的系统中有很多结构相同的表，而你可能需要同时对所有表结构进行相同的修改，那么此时只需要修改此基础数据类型即可，这一点与表继承机制很相似。

例如，如果需要为示例 6-8 中定义的 `super_users` 表增加一个电话号码字段，那么只需执行以下语句即可。

```
ALTER TYPE basic_user ADD ATTRIBUTE phone varchar(10) CASCADE;
```

一般来说，如果表依赖于某个类型，那么你就不能更改该类型的定义。`CASCADE` 修饰符凌驾于此限制之上，对所有相关表应用相同的更改。

6.2 约束机制

PostgreSQL 的约束机制是我们所接触过的数据库中最先进的，同时也是最复杂的。用户可以在创建约束时定制其各方面的属性，包括约束的名称、如何处理现有数据、级联生效条件选项、如何执行匹配算法、使用哪些索引以及在何种情况下约束可以不生效等。关于完整的约束规则，建议你查阅 PostgreSQL 官方手册中的相关内容。虽然约束机制中有大量的选项可供定制，但一般来说没那么复杂，采用默认选项就够了。本节将首先介绍关系型数据库领域耳熟能详的几个概念，包括外键约束、唯一性约束以及 check 约束，然后再介绍排他性约束。



主键约束和字段唯一性约束在表级范围内不允许出现重名。一般比较推荐的做法是把表名和字段名加入到约束的名称中，这样就不会重复。为简洁起见，下面的例子中可能不会遵循这种做法。

6.2.1 外键约束

与大多数支持引用完整性的数据库一样，PostgreSQL 遵循与其相同的约定。你可以指定级联更新和删除规则以避免出现讨厌的孤立记录。示例 6-9 中展示了如何添加外键约束。

示例 6-9　建立外键约束和相应的索引

```
SET search_path=census, public;
ALTER TABLE facts ADD CONSTRAINT fk_facts_1 FOREIGN KEY (fact_type_id)
REFERENCES lu_fact_types (fact_type_id) ❶ ON UPDATE CASCADE ON DELETE RESTRICT;
❷
CREATE INDEX fki_facts_1 ON facts (fact_type_id); ❸
```

❶ 我们在 `facts` 表和 `lu_fact_types` 表之间定义了一个外键约束关系。有了这个约束以后，如果主表 `lu_fact_types` 中不存在某 `fact_type_id` 的记录，那么从表 `fact` 中就不能插入该 `fact_type_id` 的记录。

❷ 我们定义了一个级联规则，实现了以下功能：(1) 如果主表 `lu_fact_type` 的 `fact_type_id` 字段值发生了变化，那么从表 `fact` 中相应记录的 `fact_type_id` 字段值会自动进行相应修改，以维持外键引用关系不变；(2) 如果从表 `fact` 中还存在某 `fact_type_id` 字段值的记录，那么主表 `lu_fact_type` 中相同 `fact_type_id` 字段值的记录就不允许被删除。`ON DELETE RESTRICT` 是默认行为模式，也就是说这个子句不加也可以，但为了清晰起见最好加上。

❸ PostgreSQL 在建立主键约束和唯一性约束时，会自动为相应字段建立索引，但在建立外键约束时却不会，这一点需要注意。你需要为外键字段手动建立索引，以加快关联引用时的查询速度。

外键约束对于保证数据完整性是很重要的。在比较新的 PostgreSQL 版本中，它还能起到帮助规划优化处理逻辑的作用。在 PostgreSQL 9.6 中，规划器的一个改进就是利用外键关系来推断表关联语句中谓词的可选择性，提升了许多类型查询的效率。

6.2.2 唯一性约束

主键字段的值是唯一的，但每张表只能定义一个主键，因此如果你需要保证别的字段值唯一，那么必须在该字段上建立唯一性约束或者唯一索引。建立唯一性约束时会自动在后台创建一个相应的唯一索引。与主键字段类似，建立了唯一性约束的字段可以作为外键字段被别的表引用，但它可以为空。不过请注意：建了唯一索引却没有唯一性约束的字段是可以输入空值的，而且还可以使用函数来定义。下面的例子演示了如何新建一个唯一性约束。

```
ALTER TABLE logs_2011 ADD CONSTRAINT uq UNIQUE (user_name, log_ts);
```

你可能经常会遇到仅需要保证表中部分记录行唯一的情况，PostgreSQL 不支持带筛选条件的唯一性约束，但你可以通过使用唯一性的部分索引来达到相同的目的。详情请参见 6.3.4 节。

6.2.3 check约束

check 约束能够给表的一个或者多个字段加上一个条件，表中每一行记录必须满足此条件。查询规划器也会利用 check 约束来优化执行速度，比如有些查询附带的条件与待查询表的 check 约束无交集，那么规划器会立即认定该查询未命中目标并返回。示例 6-3 中就有一个 check 约束，该约束可以告诉规划器不要试图查找不符合约束条件的记录。check 约束支持基于函数和布尔表达式的条件，因此你可以发挥创意编写出一个非常复杂的约束条件来。例如，以下 check 约束可以限制 `logs` 表中所有用户名必须都小写。

```
ALTER TABLE logs ADD CONSTRAINT chk CHECK (user_name = lower(user_name));
```

特别值得注意的一点是，当表间存在继承关系时，子表会继承父表的 check 约束，但主键、外键、唯一性这三种约束却不会继承。

6.2.4 排他性约束

传统的唯一性约束在比较算法中仅使用了“等于”运算符，即保证了指定字段的值在本表的任意两行记录中都不相等，而排他性约束机制拓展了唯一性比较算法机制，可以使用更多的运算符来进行比较运算，该类约束特别适用于解决有关时间安排的问题。

PostgreSQL 9.2 中引入了区间数据类型，该类型特别适合使用排他性约束。你可以在 depesz 站点的“Waiting for 9.2 Range Data Types”这篇文章中，找到在区间数据类型上使用排他性约束的非常好的例子。

排他性约束一般是基于 GiST 类型的索引来实现的，使用基于 B- 树算法的 GiST 多列复合索引也是可以的，不过需要先安装 `btree_gist` 扩展包才能建立这种索引。多列排他性约束的一个经典应用场景就是用于安排资源。

以下是一个使用排他约束的例子。假设你的办公场所有固定数量的会议室，各项目组在使用会议室前必须预订。示例 6-10 演示了如何避免发生预订冲突。该示例中使用了 `&&` 运算符来判定时间区段是否重叠，还使用了 `=` 运算符来判定会议室房间号是否重复，请注意观察和思考此用法。

示例 6-10　防止会议室预订冲突

```
CREATE TABLE schedules(id serial primary key, room int, time_slot tstzrange);
ALTER TABLE schedules ADD CONSTRAINT ex_schedules
EXCLUDE USING gist (room WITH =, time_slot WITH &&);
```

同唯一性约束一样，PostgreSQL 会自动为排他性约束中涉及的字段建立索引。

排他性约束适用的另一个场景是处理数组类型的数据。假设有若干房间要分配给一群人搞聚会，我们把一场聚会所使用的房间称为一个 block。方便起见，我们为每个聚会生成一条记录，但需要确保两场聚会不会共享同一个房间。建表如下：

```
CREATE TABLE room_blocks(block_id integer primary key, rooms int[]);
```

为保证每两个 block 之间不会有共享的房间，可以设置一个排他性约束来防止分配重叠。很遗憾的是，排他性约束仅对 GiST 索引类型生效，而 GiST 索引又不支持在数组上建立，所以我们先安装一个扩展包，如示例 6-11 所示。

示例 6-11　防止数组 block 重叠

```
CREATE EXTENSION IF NOT EXISTS intarray;
ALTER TABLE room_blocks
 ADD CONSTRAINT ex_room_blocks_rooms
 EXCLUDE USING gist(rooms WITH &&);
```

`intarray` 扩展包的功能是支持在整型数组（支持 `int4` 和 `int8`）上建立 GiST 索引。`intarray` 安装好以后就可以对数组类型的数据建立 GiST 了，接着就可以在整型数组数据上建立排他性约束。

6.3　索引

PostgreSQL 的索引机制功能强大、特性丰富，仅仅索引部分的内容已足够写一本大部头的书。PostgreSQL 原生支持若干种类型的索引。如果你觉得还不够，PostgreSQL 还允许你为这几种索引类型自定义新的索引运算符和修饰符以作为其功能补充。如果这样还不能满足你的要求，你可以创建自己的索引类型。

PostgreSQL 还支持在同一张表中混合搭配不同的索引类型，且预计规划器将综合考虑所有的索引。例如，在一个字段上建立 B-树索引，在旁边的字段上建立 GiST 索引，查询时两个索引都可以被用上。要更深入了解规划器对索引的选用机制，请参考 PostgreSQL 官方手册中“位图索引扫描策略”一节的内容。

普通表和物化视图上均可创建索引，但外部表不行。



同一 schema 内的索引名不能重复。

6.3.1 PostgreSQL原生支持的索引类型

要想利用好 PostgreSQL 的索引能力，需要先了解其支持的不同的索引类型以及它们各自适用的场景。PostgreSQL 原生支持的索引类型包括以下几种。

B-树索引

B-树是一种关系型数据库中常见的通用索引类型。如果你对别的索引类型不感兴趣，那么一般使用 B-树索引就可以了。有的场景下 PostgreSQL 会自动创建索引（比如创建主键约束或者唯一性约束时），那么创建出来的索引就是 B-树类型的；如果你自己创建索引时未指定索引类型，那么默认也会创建 B-树类型的索引。主键约束和唯一性约束唯一支持的后台索引就是 B-树索引。

BRIN 索引

BRIN（block range index，块范围索引）是 PostgreSQL 9.4 中引入的一种索引类型，其设计目的是针对超大表做索引，在这种表上创建 B-树索引耗费的空间过大，以至于无法全部容纳在内存中，这会导致内存和磁盘间的索引数据块换入换出，从而严重影响查询速度。BRIN 索引的思路就是把一个范围内的数据页面当作一个单元来处理，这样就可以大大压缩需要索引的目标单元数。BRIN 索引占用的空间要比 B-树索引和其他索引小得多，同时建立起来也更快。但其查询时的速度相对较慢，也不能用于确保唯一主键，另外还有一些场景页不适用该类索引。

GiST 索引

GiST（generalized search tree，通用搜索树）主要的适用场景包括全文检索以及空间数据、科学数据、非结构化数据和层次化数据的搜索。该类索引不能用于保障字段的唯一性，也就是说建立了该类型索引的字段上可插入重复值，但如果把该类索引用于排他性约束就可以实现唯一性保障。

GiST 是一种有损索引，也就是说它不存储被索引字段的值，而仅仅存储字段值的一个取样，这种取样是失真的，就像把一个盒子变成了一个多边形。这就意味着需要一个额外的查找步骤以获得真正记录的值。

GIN 索引

GIN（generalized inverted index，通用逆序索引）主要适用于 PostgreSQL 内置的全文搜索引擎以及二进制 `json` 数据类型。其他一些扩展包（比如 `hstore` 和 `pg_trgm`）也会使用这种索引。GIN 其实是从 GiST 派生出来的一种索引类型，但它是无损的，也就是说索引中会包含有被索引字段的值。如果你需要查询的字段都已被索引，那么只读取索引即可获取查询结果，这种情况下 GIN 的查询速度是快于 GiST 的。然而，由于 GIN 比 GiST 在更新操作时要多出一个字段值复制动作，因此此时 GIN 索引体积更大并且更新速度慢于 GiST 索引。另外，GIN 的索引树内部每一个索引行的长度是有限制的，所以它不能用于对 hstore 文档或者 `text` 等大对象类型进行索引。如果你需要把一个 600 页的手册内容存入一张表的某个字段，那么绝对不要在该字段上建立 GIN 类型的索引。在“Waiting for Faster LIKE/ILIKE 符”这篇文章中有一个关于 GIN 用法的非常好的例子可供你参考。在 9.3 版中，用于实现字符串模糊匹配和相似度查询的 `pg_trgm` 扩展包中做了一个功能强化：支持正则表达式条件查询时用上 GIN 索引，这大大增加了 `pg_trgm` 的适用场景。

SP-GiST 索引

SP-GiST 是指基于空间分区树（space-partitioning trees）算法的 GiST 索引。该类型的索引与 GiST 索引的适用领域相同，但对于某些特定领域的数据算法，其效率会更高一些。PostgreSQL 的 `point` 和 `box` 等原生几何类型以及 `text` 类型是最先支持该类索引的数据类型。从 9.3 版开始，区间类型也开始支持此类型的索引。

散列索引

散列索引在 GiST 和 GIN 索引出现前就已经得到了广泛使用。业界普遍认为 GiST 和 GIN 索引在性能和事务安全性方面要胜过散列索引。PostgreSQL 10 之前的版本中，事务日志中不会记录散列索引的变化，那么在流式复制环境中就不能使用散列索引，否则会导致修改无法被同步。尽管有一段时期散列索引被 PostgreSQL 官方列为不推荐使用状态，但是在 PostgreSQL 10 中它再次得到了强化。该版本中，散列索引强化了事务一致性以及一些性能提升，有的场景中它会比 B-树更快。

基于 B-树算法的 GiST 和 GIN 索引

如果你想了解 PostgreSQL 除了原生索引以外还有哪些索引，不管是出于业务需要还是仅仅出于好奇，都可以从了解基于 B-树算法的 GiST 和 GIN 索引开始。二者都可以用扩展包形式安装，并且大多数 PostgreSQL 发行版中都含有这两个扩展包。这两类混合算法索引的优势在于，它们既能够支持 GiST 和 GIN 索引特有的运算符，又具有 B-树索引对于“等于”运算符的良好支持。当需要建立同时包含简单和复杂数据类型的多列复合索引时，你会发现这两类索引不可或缺。例如，我们建立的复合索引中既有普通文本类型也有 full-text 类型[1]。一般来说，类似全文检索、`ltree`、`geometric` 和空间类型这些高级数据类型，只能使用 GIN 或者 GiST 索引，因此这类字段不可能与只能建立 B-树索引的普通字段构成复合索引。此时基于这两种索引就可以实现这个目标，基于它们的混合算法可以把建立了 GiST 索引的字段和建立了 B-树索引的字段联合起来，组成为单一的复合索引。

除了 PostgreSQL 原生附带的索引类型外，还有一些额外的索引类型以扩展包形式存在。其中比较流行的是 VODKA 和 RUM（基于 GIN 索引的一个变种）这两种，适用于 PostgreSQL 9.6 以及之后的版本。RUM 索引适用于 full-text 之类的复杂类型，如果你需要全文词组搜索，那么就必须使用 RUM 类型。另外，它还提供了一些距离运算符。

另一个索引类型是 pgroonga，它以扩展包的形式存在，当前仅支持 PostgreSQL 9.5 和 PostgreSQL 9.6。该扩展把作为全文搜索引擎同时也是一个列式存储容器的 roonga 的能力带入了 PostgreSQL。PGRoonga 扩展包中含有一个名为 pgroonga 的索引类型以及相应的运算符。PGRoonga 扩展可以实现对普通文本进行索引，以支持对其进行全文检索，但不像 PostgreSQL 原生的全文检索机制那样需要创建全文检索向量。PGRoonga 还能够实现让 `ILIKE` 和 `LIKE '%something%'` 这种操作用上索引，效果类似于 `pg_trgm` 扩展。此外，它还支持对数组类型和 JSONB 类型建立索引。Linux/Unix 和 Windows 下该扩展都有可用的安装包。

注 1：PostgreSQL 没有 full-text 类型这种说法，此处作者指的是 `tsvector` 和 `tsquery` 这两种专用于全文检索的数据类型。——译者注

6.3.2 运算符类

大多数人即使不知道“运算符类”是什么以及它与索引有什么关系，也能毫无障碍地使用索引。但如果你运气没这么好，也就是说你的应用场景需要你了解这些内容，那么就得好好研究运算符类了，否则就会一直被这个问题困扰：“为什么规划器没用上我的索引？”

各种数据类型均有自身的特点，因此适用的索引类型不同，会用到的比较运算符也不同。例如，对于基于区间类型（range）的索引来说，最常用的运算符是重叠运算符（`&&`），然而该运算符对于本文搜索领域来说却毫无意义。对于中文这类表意文字来说，建立的索引基本上不会用到“不等于”运算符；而对英文这类表音文字建立索引时，字母 A 到 Z 的排序操作是不可或缺的。

基于以上特点，PostgreSQL 把一类应用领域相近的运算符以及这些运算符适用的数据类型组合在一起，称为一个运算符类。例如，`int4_ops` 运算符类包含适用于 `int4`（也就是日常所说的 `integer`）类型的 `= < > > <` 运算符。PostgreSQL 提供了一张叫作 `pg_class` 的系统表，从中可以查到完整的运算符类列表，其中既包含了系统原生支持的类，也包含了通过扩展包机制添加的类。一种类型的索引会使用特定的若干种运算符类。完整的运算符列表可以从 pgAdmin 界面上的运算符类分支下看到，也可以根据 `system` catalog 在示例 6-12 中执行查询得到。

示例 6-12　查询 B-树索引支持的数据类型以及运算符类

```
SELECT am.amname AS index_method, opc.opcname AS opclass_name,
opc.opcintype::regtype AS indexed_type, opc.opcdefault AS is_default
FROM pg_am am INNER JOIN pg_opclass opc ON opc.opcmethod = am.oid
WHERE am.amname = 'btree'
ORDER BY index_method, indexed_type, opclass_name;

index_method | opclass_name        | indexed_type | is_default
-------------+---------------------+--------------+-----------
btree        | bool_ops            | boolean      | t
⋮
btree        | text_ops            | text         | t
btree        | text_pattern_ops    | text         | f
btree        | varchar_ops         | text         | f
btree        | varchar_pattern_ops | text         | f
⋮
```

在示例 6-12 中，仅查询了 B-树的相关数据。请注意，每类索引都会有多个运算符类，而其中仅有一个会被标记为默认运算符类。如果建立索引时未指定使用哪个运算符类，那么 PostgreSQL 会使用默认运算符类。绝大多数情况下这么做是没什么问题的，但并非绝对如此。

例如，B-树索引默认的 `text_ops` 运算符类（又名 `varchar_ops`）中并不支持 ~~ 运算符（即 `LIKE` 运算符），所以如果创建 B-树索引时选择了该运算符类，那么所有使用 `LIKE` 的查询都无法在 `text_ops` 运算符类中使用索引。因此，如果你的业务场景需要对 `varchar` 或者 `text` 类型进行大量 `LIKE` 模糊查询，那么创建索引时最好显式指定使用 `text_pattern_ops` 或者 `varchar_pattern_ops` 这两个运算符类。指定运算符类的语法很简单，只需要在创建索引时加在被索引字段名的后面即可，参考示例如下。

```
CREATE INDEX idx1 ON census.lu_tracts USING btree (tract_name text_pattern_ops);
```



在示例 6-12 的查询结果中，你可能已经注意到了 B-树索引的 varchar_ops 和 text_ops 两种运算符类适用的数据类型都是 text，这样来看的话，varchar_ops 貌似有点名不副实。这是因为 varchar 类型本质上就是加了长度限制的 text 类型，二者可以共用一套运算符。varchar_ops 和 varchar_pattern_ops 实质上就是 text_ops 和 text_pattern_ops 的别名，之所以把前面两种单列出来是因为 varchar 毕竟是单独的一种数据类型，存在一种以其类型命名的专属运算符类看起来会比较合理。

最后请牢记这一条：你创建的每一个索引都只会使用一个运算符类。如果希望一个字段上的索引使用多个运算符类，那么请创建多个索引。要将默认索引 text_ops 添加到表中，请运行以下代码：

```
CREATE INDEX idx2 ON census.lu_tracts USING btree (tract_name);
```

现在，在同一个字段上就有了多个索引（单个字段上可建立的索引个数是没有限制的）。规划器处理等值查询时会使用 idx2，处理 like 模糊查询时会使用 idx1。

你可以在 PostgreSQL 官方手册中找到关于运算符类的更详细的描述。另外，我们也强烈建议你阅读一下我们的博客文章“Why is My Index Not Used?”。

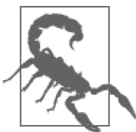
6.3.3　函数索引

PostgreSQL 的函数索引功能可以基于字段值的函数运算结果建立索引。函数索引的用途也是很广泛的，例如可用于对大小写混杂的文本数据建立索引。PostgreSQL 是一个区分大小写的数据库，如果要实现不区分大小写的查询，可以借助如下的函数索引：

```
CREATE INDEX idx ON featnames_short
USING btree (upper(fullname) varchar_pattern_ops);
```

在下面的查询语句中，我们使用了前面建立索引时使用的 upper 函数来将 fullname 字段变为大写后，再进行条件比较。由于查询语句中使用的函数与我们在建立索引时使用的函数相同，因此规划器会对此查询语句使用索引：

```
SELECT fullname FROM featnames_short WHERE upper(fullname) LIKE 'S%';
```



注意，查询语句中使用的函数要与建立函数索引时使用的函数完全一致，这样才能保证用上索引。

6.3.4　基于部分记录的索引

基于部分记录的索引（有时也称为已筛选索引）是一种仅针对表中部分记录的索引，而且这部分记录需要满足 WHERE 语句设置的筛选条件。例如，假设某表中有 1 000 000 条记录，但你只会查询其中一个记录数为 10 000 的子集，那么该场景就非常适合使用基于部分记录的索引。这种索引比全量索引快，因为其体积小，所以可以把更多索引数据缓存到内存

中。另外，该类索引占用的磁盘空间也更小。

基于部分记录的索引能够实现仅针对部分记录的唯一性约束。举个例子，假设你手上有一家报纸在过去 10 年间的订阅用户数据，现在需要确保还在订阅的用户不会每天多拿一份报纸。由于人们对纸媒的兴趣下降，因此这 10 年间的全量订阅用户中仅有 5% 还在坚持订阅。所以，很显然你不需要关注那些已经退订的用户，因为他们的姓名早已从报纸递送员手上的递送名单中剔除。表结构如下：

```
CREATE TABLE subscribers (
    id serial PRIMARY KEY,
    name varchar(50) NOT NULL, type varchar(50),
    is_active boolean);
```

我们建立一个基于当前活跃用户的部分记录索引即可：

```
CREATE UNIQUE INDEX uq ON subscribers USING btree(lower(name)) WHERE is_active;
```



索引的 WHERE 条件中使用的函数必须是确定性函数，即固定的输入一定能够得到固定输出的函数。这意味着有几类函数是不能用作筛选条件的：一类是 CURRENT_DATE 这种输出结果不停在变的函数；一类是依赖于其他表数据进行运算的函数，其输出结果受其他表的数据的影响，因此输出也是不固定的；还有一类是依赖当前表中的其他记录行进行运算的函数，其输出也不会受控。

需要特别强调的一点是，当使用 SELECT 语句查询数据时，要想规划器用上部分记录索引，那么该查询语句的 WHERE 条件中必须包含创建部分记录索引时所使用的 WHERE 条件；如果该索引同时还是个函数索引，那么该查询语句的 WHERE 条件中必须包含建立索引时所使用的函数。前面例子中创建的部分数据索引同时也是个函数索引，因为条件中使用了 lower 函数而不是仅使用了 name 字段。这个逻辑实际操作起来比较麻烦也容易出错，有一个办法可以让事情变得简单一些，那就是建立一个视图，视图条件就是建立索引的条件，那么针对此视图进行查询就永远不会漏掉条件了。还是以前述报纸订阅用户数据为例，建立如下视图：

```
CREATE OR REPLACE VIEW vw_subscribers_current AS
SELECT id, lower(name) As name FROM subscribers WHERE is_active = true;
```

然后将针对原表的查询都改为针对此视图的查询即可（一种比较激进的观点认为，此种情况下永远都不应该直接查询原表）。视图的本质就是一个保存下来的查询语句，创建视图时所用的 WHERE 条件以及其中的函数条件部分都会被完整代入到任何针对该视图的查询语句中。当然查询视图的语句也可以包含额外的查询条件，这些过滤条件最后都会生效。针对前面我们刚刚创建的视图来说，查询视图的语句通过两个代入步骤后即可被翻译为针对基表的查询语句，之后就可以用上基础表的部分数据索引。这两个步骤分别为：首先把查询语句中的 name 字段映射到基础表的 lower(name)，这样查询视图的 name 字段就等同于查询视图基础表的 lower(name)；然后把创建视图时的 is_active=true 条件添加到原始的视图查询语句中。这样代入后的语句就可以用上基础表的部分数据索引了：

```
SELECT * FROM vw_subscribers_current WHERE name = 'sandy';
```

你可以查看规划器输出的执行计划以确认你的索引是否被用上了。

6.3.5 多列索引

在本章前面的内容中，你应该已经见到了大量多列索引（也称为复合索引[2]）的例子。请注意，也可以基于多个字段来创建函数索引。以下是一个多列索引的示例。

```
CREATE INDEX idx ON subscribers
USING btree (type, upper(name) varchar_pattern_ops);
```

PostgreSQL 的规划器在语句执行过程中会自动使用一种被称为“位图索引扫描”的策略来同时使用多个索引。该策略可以使得多个单列索引同时发挥作用，达到的效果与使用单个复合索引相同。如果你不能确定业务的应用模式是以单列作为查询条件的场景多一些，还是同时以多列作为查询条件的场景多一些，那么最好针对可能作为查询条件的每个列单独建立索引，这是最灵活的做法，规划器会决定如何组合使用这些索引。

假设你建了一个 B- 树多列索引，其中包含 `type` 和 `upper(name)` 两个字段，那么完全没必要针对 `type` 字段再单独建立一个索引，因为规划器即使在遇到只有 `type` 单字段的查询条件时，也会自动使用该多列索引，这是规划器的一项基本能力。如果查询条件字段没有从多列索引中的第一个字段开始匹配，规划器其实也能用上索引，但请尽量避免这种情况，因为从索引原理上说，从索引的第一个字段开始匹配才是最高效的。

规划器支持一种仅依赖索引内数据的查询策略（index-only scan），也就是说如果查询的目标字段在索引内都有，那么直接扫描索引就可以得到查询结果，根本不需要访问表的本体了。这个功能的引入使得复合索引的作用更为凸显，因为复合索引可以提供更多数据，因此更适合使用此种查询方法。如果你的业务场景中查询的目标字段和条件字段是相同的那几个，那么就应该建立复合索引以提升查询速度。不过，索引中包含的字段越多也就意味着索引占用的空间会越大，能在内存中缓存的索引条目就越少，因此请不要滥用复合索引。

注 2：复合索引的多个字段中也可以包含函数，此时建立的索引既是复合索引也是函数索引。——译者注

第 7 章

PostgreSQL的特色SQL语法

PostgreSQL 在对 ANSI SQL 标准的遵从方面已经远远走在了其他数据库的前面。除了适配标准之外，PostgreSQL 还提供了类型多样的强化语法，这些强化包括易用的简化语法以及一些前卫到足以打破关系型 SQL 边界的语法特性。通过这些努力，PostgreSQL 保持了领先地位。本章将介绍一些其他数据库中很少见的 SQL 语法特性。希望你已有一定的 SQL 开发经验，否则可能无法理解 PostgreSQL 为简化 SQL 开发工作所做的努力。

7.1 视图

一个设计良好的关系型数据库中的表存储的是规范化的数据，因此当需要从多张表中获取数据时，就需要写关联查询的 SQL 语句。如果你的应用场景需要反复执行这种关联查询语句，可以考虑创建一个视图。简单来说，视图就是持久化存储在数据库中的一个查询语句。

有人认为用户不应该直接访问表，而应该通过视图来访问，不过这就意味着需要为每张表都创建一个视图。这种做法的优点是在表的本体之上增加了一个访问层，简化了权限管理且使得业务逻辑抽象更容易，缺点就是太麻烦。我们认为这个观点是合理的，但事实上由于惰性很少有人会这么做。

PostgreSQL 的视图功能近年来有了长足的发展。9.3 版中推出了可自动更新的视图。如果你的视图是基于单表的，并且视图字段中包含了基础表的主键字段，那么就可以直接对此视图执行 `UPDATE` 操作，视图的基础表数据将随之更新。

9.3 版中还引入了物化视图。每个视图都对应一个 SQL 查询语句，视图本质上就是该 SQL 的查询结果集的一个别名。每次访问视图时都需要执行其对应的 SQL，但物化视图将视图逻辑映射后的数据记录实际存储下来，这样访问物化视图时就省略了视图底层 SQL 的执行过程，就像访问一张本地表一样。一旦物化视图建立好以后，只有对它执行 `REFRESH` 操作

时才会再次从基础表中读取数据。根据前面的描述可以知道，使用物化视图可以节省计算资源，因为视图底层 SQL（这种 SQL 逻辑可能极其复杂）不用反复执行。但物化视图也有缺点，因为如果刷新不及时，就会导致取出的数据可能不是最新的，还有一个问题就是物化视图刷新期间会不可访问。

9.4 版开始支持用户在物化视图刷新时也能对其进行访问，该版本还引入了 `WITH CHECK OPTION` 修饰符，用于防止在视图的范围之外进行插入和更新。

7.1.1 单表视图

最简单的视图是从单个表得出的。如果打算将数据写回到该表，请始终包含主键，如示例 7-1 所示。

示例 7-1 创建基于单表的视图

```
CREATE OR REPLACE VIEW census.vw_facts_2011 AS
SELECT fact_type_id, val, yr, tract_id FROM census.facts WHERE yr = 2011;
```

自从 9.3 版起，就可以使用 `INSERT`、`UPDATE` 和 `DELETE` 命令在该视图中更改数据了。更新和删除命令将遵从作为视图一部分的任何 `WHERE` 条件。例如，下面的删除命令将仅删除 `val` 字段值为 0 的记录：

```
DELETE FROM census.vw_facts_2011 WHERE val = 0;
```

以下 `UPDATE` 操作不会更新任何记录，因为视图仅包含 2011 年的数据：

```
UPDATE census.vw_facts_2011 SET val = 1 WHERE yr = 2012;
```

请注意，针对视图插入的数据可能不会被包含在视图可见范围内；针对视图的更新操作也会发生类似现象，即更新后的数据不再被包含在视图的可见范围内，如示例 7-2 所示。

示例 7-2 针对视图进行 `UPDATE` 操作，修改后的数据不再落在视图可见范围内

```
UPDATE census.vw_facts_2011 SET yr = 2012 WHERE yr = 2011;
```

示例 7-2 中的 `UPDATE` 语句操作的目标记录是落在视图可见范围内的，但更新后会把本来落在视图可见范围内的记录变成落到视图可见范围之外，也就是说视图更新后少了一条记录。但为了保持视图数据的一致性，我们不希望这种情况发生，也就是说希望更新后的数据仍然应该落在视图可见范围内，因此任何会导致这种情况发生的插入或者更新操作都应被禁止。这可以通过 9.4 版中引入的 `WITH CHECK OPTION` 子句来实现。创建视图时如果附带了此子句，那么此视图中插入的数据或者更新后的数据落在视图可见范围之外时，系统会报错，违反了该约束的操作会失败。下面我们将限定 `vs_facts_2011` 视图仅允许插入 2011 年的数据，同时不允许将 `yr` 字段修改为 2011 以外的其他值。我们修改一下视图定义，把这个约束加上，语法如示例 7-3 所示。

示例 7-3 创建带有 `WITH CHECK OPTION` 约束的单表视图

```
CREATE OR REPLACE VIEW census.vw_facts_2011 AS
SELECT fact_type_id, val, yr, tract_id FROM census.facts
WHERE yr = 2011 WITH CHECK OPTION;
```

尝试执行以下更新操作：

```
UPDATE census.vw_facts_2011 SET yr = 2012 WHERE val > 2942;
```

你会看到这样的报错信息：

```
ERROR: new row violates WITH CHECK OPTION for view"vw_facts_2011"
DETAIL: Failing row contains (1, 25001010500, 2012, 2985.000, 100.00).
```

7.1.2　使用触发器来更新视图

视图可以将针对多张表的关联查询封装为针对视图的简单查询。如果视图的基础表有多张，那么直接更新该视图是不允许的，因为多张表必然带来的问题就是操作要落到哪个基础表上，PostgreSQL 是无法自动判定的。假设你有一个视图，该视图基于一张国家信息表和一张省份信息表，此时你希望删除该视图的一条记录，PostgreSQL 无法得知你到底想要仅删除一条国家记录，还是仅删除一个省份记录，抑或是删除一个国家以及该国家对应的所有省的记录。PostgreSQL 无法自动判定你想做什么并不代表就不能对这种复杂视图进行修改操作，你可以通过编写触发器来对这些操作进行转义处理，转义后的逻辑中可以体现你的意图。

我们首先建立一个关联了两张表的视图，如示例 7-4 所示。

示例 7-4　创建 vw_facts 视图

```
CREATE OR REPLACE VIEW census.vw_facts AS
SELECT
        y.fact_type_id, y.category, y.fact_subcats, y.short_name,
        x.tract_id, x.yr, x.val, x.perc
FROM census.facts As x INNER JOIN census.lu_fact_types As y
ON x.fact_type_id = y.fact_type_id;
```

要实现通过触发器来更新视图这一目标，需要定义一个或者多个 INSTEAD OF 触发器来实现针对 INSERT、UPDATE、DELETE 这三大基本操作的转义处理。值得一提的是，PostgreSQL 支持基于 TRUNCATE 事件的触发器。触发器需要有一个基础函数，你可以使用除 SQL 外的任何语言来编写该基础函数，其命名也没有规则限制。我们在示例 7-5 中选择使用 PL/pgSQL 语法来编写。

示例 7-5　在 vw_facts 视图上创建一个对 INSERT、UPDATE、DELETE 操作进行转义处理的函数

```
CREATE OR REPLACE FUNCTION census.trig_vw_facts_ins_upd_del() RETURNS  trigger AS
$$
BEGIN
    IF (TG_OP = 'DELETE') THEN ❶
        DELETE FROM census.facts AS f
        WHERE
            f.tract_id = OLD.tract_id AND f.yr = OLD.yr AND
            f.fact_type_id = OLD.fact_type_id;
        RETURN  OLD;
    END IF;
    IF (TG_OP = 'INSERT') THEN ❷
        INSERT INTO census.facts(tract_id, yr, fact_type_id, val, perc)
        SELECT NEW.tract_id, NEW.yr, NEW.fact_type_id, NEW.val, NEW.perc;
        RETURN NEW;
    END IF;
```

```
        IF (TG_OP = 'UPDATE') THEN ❸
            IF
                ROW(OLD.fact_type_id, OLD.tract_id, OLD.yr, OLD.val, OLD.perc) !=
                ROW(NEW.fact_type_id, NEW.tract_id, NEW.yr, NEW.val, NEW.perc)
            THEN ❹
                UPDATE census.facts AS f
                SET
                    tract_id = NEW.tract_id,
                    yr = NEW.yr,
                    fact_type_id = NEW.fact_type_id,
                    val = NEW.val,
                    perc = NEW.perc
                WHERE
                    f.tract_id = OLD.tract_id AND
                    f.yr = OLD.yr AND
                    f.fact_type_id = OLD.fact_type_id;
                RETURN NEW;
            ELSE
                RETURN NULL;
            END IF;
        END IF;
    END;
    $$
    LANGUAGE plpgsql VOLATILE;
```

❶ 对删除操作进行转义处理，筛选条件的字段取值来源于 `OLD` 记录。[1]

❷ 对插入操作进行转义处理。

❸ 对更新操作进行转义处理。根据 `OLD` 记录的内容判断哪些记录要更新为 `NEW` 记录。[2]

❹ 比较 `OLD` 记录和 `NEW` 记录的字段值，只有二者不一样时才真正执行更新动作。

接下来，我们将此触发器函数绑定到视图上，语法如示例 7-6 所示。

示例 7-6　将触发器函数绑定到视图上

```
CREATE TRIGGER trig_01_vw_facts_ins_upd_del
INSTEAD OF INSERT OR UPDATE OR DELETE ON census.vw_facts
FOR EACH ROW EXECUTE PROCEDURE census.trig_vw_facts_ins_upd_del();
```

可以看到，这个绑定过程所使用的语法几乎就是一行很自然的英文句子，这种情况在 SQL 领域还是不多见的。

现在针对视图进行更新、删除或插入操作时，这些操作将更新基础 `facts` 表：

```
UPDATE census.vw_facts SET yr = 2012
WHERE yr = 2011 AND tract_id = '25027761200';
```

执行成功后 PostgreSQL 会输出以下信息：

```
Query returned successfully: 56 rows affected, 40 ms execution time.
```

注 1：OLD 记录是指原始的针对视图的删除动作所要删除的视图记录。也就是说，OLD 记录是视图记录，而非视图基础表的记录。——译者注

注 2：NEW 记录指的是原始的针对视图的更新动作设置的修改后的视图记录。也就是说，NEW 记录是视图记录，而非视图基础表的记录。——译者注

如果试图更新的字段不在前面触发器函数中所列的可更新字段列表中，那么更新操作就不会命中任何记录：

```
UPDATE census.vw_facts SET short_name = 'test';
```

输出消息将如下所示：

```
Query returned successfully: 0 rows affected, 931 ms execution time.
```

前面的例子中我们用一个触发器函数处理了所有类型的触发事件（INSERT、UPDATE、DELETE），但实际上专门为每种触发事件创建一个独立的触发器也是可以的。

PostgreSQL 还支持一种名为 rules 的规则机制来更新视图，该机制的出现早于 INSTEAD OF 触发器支持视图这一功能。博文“Database Abstraction with Updateable Views”中有基于 rules 机制实现可更新视图的例子。

你依然可以使用 rules 机制来实现视图数据的更新，但我们更推荐使用 INSTEAD OF 触发器机制。从内部实现机制看，PostgreSQL 依然使用了 rules 机制来实现视图功能（视图本质上就是一个 INSTEAD OF SELECT 规则加上一个虚拟表）和基于单表的可更新视图功能。rules 机制和触发器机制的区别在于：rules 机制通过重写原始 SQL 达成目标；触发器通过拦截每行操作并改变其实现逻辑来达成目标。可以看到，当一个操作涉及多张表时，rules 系统无论是编写还是理解起来都比触发器困难得多。rules 机制还有一个缺陷，就是只能用 SQL 语言编写，其他过程式语言都不支持。

7.1.3　物化视图

物化视图会把视图可见范围内的数据在本地缓存下来，然后就可以当成一张本地表来使用。首次创建物化视图以及对其执行 REFRESH MATERIALIZED VIEW 刷新操作时，都会触发数据缓存动作，只不过前者是全量缓存，后者是增量刷新。请注意，物化视图特性是从 9.3 版开始支持的。

物化视图最典型的应用场景是用于加速时效性要求不高的长时复杂查询，在 OLAP（在线分析与处理）领域，这种查询经常出现。

物化视图的另一个特点就是支持建立索引以加快查询速度。示例 7-7 建立了示例 7-1 中的视图的物化版本。

示例 7-7　建立物化视图

```
CREATE MATERIALIZED VIEW census.vw_facts_2011_materialized AS
SELECT fact_type_id, val, yr, tract_id FROM census.facts WHERE yr = 2011;
```

然后对物化视图建立一个索引，语法与在普通表上建立索引完全相同，如示例 7-8 所示。

示例 7-8　在物化视图上建立索引

```
CREATE UNIQUE INDEX ix
ON census.vw_facts_2011_materialized (tract_id, fact_type_id, yr);
```

当物化视图中含大量记录时，为了加快对它的访问速度，我们需要对数据进行排序。要实现这一点，最简单的方法就是在创建物化视图时使用的 SELECT 语句中增加 ORDER BY 子句。

另外一种方法就是对其执行聚簇排序操作，以使得记录的物理存储顺序与索引的顺序相同，具体步骤是：首先创建一个索引，该索引应体现你所希望的排序；然后基于指定索引对物化视图执行 CLUSTER 命令，语法如示例 7-9 所示。

示例 7-9　基于某索引对物化视图执行聚簇排序操作

```
CLUSTER census.vw_facts_2011_materialized USING ix; ❶
CLUSTER census.vw_facts_2011_materialized; ❷
```

❶ 指定聚簇操作所依据的索引名。执行过以后，系统就会自动记下该表是依据哪个索引进行聚簇排序的，后面再次执行聚簇操作时系统会自动使用该索引，所以索引名仅在首次聚簇操作时需要，后续不再需要。

❷ 每次刷新过物化视图后，都需要重新对其进行一次聚簇排序操作。

相对于 CLUSTER 方案来说，ORDER BY 方案的优点在于，每次执行 REFRESH MATERIALIZED VIEW 时都会自动对记录进行重排序，但 CLUSTER 方案就必须手动执行；其缺点在于物化视图加了 ORDRE BY 以后，REFRESH 操作执行会耗时更久。在正式使用 ORDER BY 方案之前，你应该对 REFRESH 操作进行测试，看其性能是否可接受。另一种测试方法是直接运行创建物化视图时所用的带 ORDER BY 的 SQL 语句。

在 PostgreSQL 9.3 中，刷新物化视图的语法如下：

```
REFRESH MATERIALIZED VIEW census.vw_facts_2011_materialized;
```

在 PostgreSQL 9.4 中，为了解决物化视图刷新操作时不可访问的问题，可以使用以下语法：

```
REFRESH MATERIALIZED VIEW CONCURRENTLY census.vw_facts_2011_materialized;
```

物化视图有以下几个缺点。

- 不支持通过 CREATE OR REPLACE 语法来对一个现有的物化视图进行重建，要想重建只能先删除再重建，即使只做很小的改变也需要这么做。删除语法是 DROP MATERIALIZED VIEW + *视图名*。删除以后，该视图上所有的索引都会丢失。
- 每次刷新数据时都要执行一次 REFRESH MATERIALIZED VIEW 操作。PostgreSQL 不支持自动刷新物化视图。要想实现自动刷新，必须使用类似 crontab、pgAgent 定时任务或者是触发器之类的机制。博文“Caching Data-with Materialized Views and Statement-Level-Trigger”中提供了一个使用触发器来进行刷新的例子，可供你参考。
- 在 9.3 版中，刷新物化视图是一个阻塞操作；也就是说，视图在刷新期间是无法访问的。9.4 版中引入了一个新的 CONCURRENTLY 关键字，在 REFRESH 命令中增加了该关键字以后，可以解除锁定限制，前提条件是被刷新的物化视图上要有一个唯一索引。实现无锁刷新的代价就是，如果有人在刷新期间进行视图访问，那么刷新时间会变长。

7.2　灵活易用的PostgreSQL专有SQL语法

我们在多年编写 SQL 语句的过程中使用过很多 PostgreSQL 专有语法，借助它们可以编写出更加简洁以及功能更加强大的 SQL。本节介绍的语法都是 PostgreSQL 的专有语法。“专有”意味着该语法不符合 ANSI SQL 标准。如果你的老板要求你编写的 SQL 必须遵守 ANSI SQL 标准，那么请不要使用本节介绍的专有语法。

7.2.1 DISTINCT ON

我们认为最好用的一个专有语法就是 DISTINCT ON，其功能类似于 DISTINCT，但可以精确到更细的粒度。DISTINCT 会将结果集中完全重复的记录剔除，但 DISTINCT ON 可以将结果集中指定字段值的重复记录剔除，具体实现方法是先对结果集按照 DISTINCT ON 指定的字段进行排序，然后筛选出每个字段值第一次出现时所在的记录，其余的记录都剔除。可以看到，一个小小的单词 ON 就实现了必须写大量代码才能实现的功能。

示例 7-10 中演示了如何获取马萨诸塞州每个县的第一个人口统计区的信息。

示例 7-10　DISTINCT ON 的用法

```
SELECT DISTINCT ON (left(tract_id, 5))
    left(tract_id, 5) As county, tract_id, tract_name
FROM census.lu_tracts
ORDER BY county, tract_id;

county |  tract_id   | tract_name
-------+-------------+---------------------------------------------------
25001  | 25001010100 | Census Tract 101, Barnstable County, Massachusetts
25003  | 25003900100 | Census Tract 9001, Berkshire County, Massachusetts
25005  | 25005600100 | Census Tract 6001, Bristol County, Massachusetts
25007  | 25007200100 | Census Tract 2001, Dukes County, Massachusetts
25009  | 25009201100 | Census Tract 2011, Essex County, Massachusetts
:
(14 rows)
```

请注意，ON 修饰符支持设置多列，运算时将基于这多个列的总体唯一性来进行去重操作。同时，查询语句中 ORDER BY 子句的排序字段列表的最左侧必须是 DISTINCT ON 指定的字段列表，即保证整个结果集是按照这几个字段排序的，这样最终去重后得到的结果才是你想要的。

7.2.2 LIMIT和OFFSET关键字

LIMIT 关键字指定了查询时仅返回指定数量的记录，OFFSET 关键字指定了从第几条记录开始返回。你可以将二者结合起来使用，也可以单独使用。一般来说，这两个关键字总是和 ORDER BY 联用的，因为只有在一个已经按照用户的意图排好序的结果集上指定返回特定的子结果集才有意义。示例 7-11 中演示了 OFFSET 关键字的用法，如果不设置 OFFSET 的话，其值默认为 0。

该语法并非 PostgreSQL 所特有，事实上它最早源自于 MySQL。这种限制返回结果记录数的功能在很多数据库中都支持，但具体语法和内部实现机制各有千秋。

示例 7-11　要求示例 7-10 的查询结果集仅返回从第 3 条开始的 3 条记录

```
SELECT DISTINCT ON (left(tract_id, 5))
    left(tract_id, 5) As county, tract_id, tract_name
FROM census.lu_tracts
ORDER BY county, tract_id LIMIT 3 OFFSET 2;

county | tract_id    | tract_name
```

```
-------+-------------+------------------------------------------------
25005  | 25005600100 | Census Tract 6001, Bristol County, Massachusetts
25007  | 25007200100 | Census Tract 2001, Dukes County, Massachusetts
25009  | 25009201100 | Census Tract 2011, Essex County, Massachusetts
(3 rows)
```

7.2.3 简化的类型转换语法

ANSI SQL 标准中定义了一个名为 CAST 的类型转换函数，可以实现数据类型之间的互转。例如 CAST('2011-1-1' AS date) 可以将文本 2011-1-1 转换为一个日期型数据。PostgreSQL 支持一种简写语法，该语法使用了两个冒号来表示转换关系，具体格式为：'2011-1-1'::date。这种写法形式更简洁也更易于使用，比如有时候需要级联执行多个类型转换动作，也就是说需要将类型 A 转换为类型 B 再转换为类型 C，这种情况用简写语法也是可以实现的，例如 someXML::text::integer。

7.2.4 一次性插入多条记录

PostgreSQL 支持一次性插入多条记录的语法。示例 7-12 演示了如何向示例 6-3 中创建的表中一次性插入多条记录。

示例 7-12　一次性插入多条记录

```
INSERT INTO logs_2011 (user_name, description, log_ts)
VALUES
    ('robe', 'logged in', '2011-01-10 10:15 AM EST'),
    ('lhsu', 'logged out', '2011-01-11 10:20 AM EST');
```

请注意，在 PostgreSQL 中 VALUES 子句并不是只能作为 INSERT 语句的一部分来使用，它其实是一个动态生成的临时结果集，可用于多种场合，如示例 7-13 所示。

示例 7-13　使用 VALUES 语法来模拟一个虚拟表

```
SELECT *
FROM (
    VALUES
        ('robe', 'logged in', '2011-01-10 10:15 AM EST'::timestamptz),
        ('lhsu', 'logged out', '2011-01-11 10:20 AM EST'::timestamptz)
    ) AS l (user_name, description, log_ts);
```

将 VALUES 子句当作一个虚拟表来用时，需要为该表指定字段名，并对那些无法隐式转换的字段值显式地进行类型转换。MySQL 和 SQL Server 也支持该语法。

7.2.5 使用ILIKE实现不区分大小写的查询

PostgreSQL 是一套区分大小写的系统，但它也可以实现不区分大小写的文本搜索，实现方法有两种。一种是将 ANSI LIKE 运算符两边的文本都用 upper 函数转为大写，但这样会导致用不上索引，或者必须单独建立一个基于 upper 函数的函数式索引才能使查询语句用上索引。另一种是使用 PostgreSQL 所特有的 ILIKE 运算符（~~*），语法如下：

```
SELECT tract_name FROM census.lu_tracts WHERE tract_name ILIKE '%duke%';
```

```
tract_name
------------------------------------------------
Census Tract 2001, Dukes County, Massachusetts
Census Tract 2002, Dukes County, Massachusetts
Census Tract 2003, Dukes County, Massachusetts
Census Tract 2004, Dukes County, Massachusetts
Census Tract 9900, Dukes County, Massachusetts
```

7.2.6 使用ANY运算符进行数组搜索

PostgreSQL 有一个专用于数组数据处理的 ANY 运算符，一般用于单个值与数组元素值的匹配比较场景，所以使用时还会需要有一个比较运算符或者比较关键字。ANY 运算符的效果是，只要数组中的任何元素与该行被比较的字段值匹配，则该行记录就被查询条件命中。

以下是一个例子：

```
SELECT tract_name
FROM census.lu_tracts
WHERE tract_name ILIKE ANY(ARRAY['%99%duke%','%06%Barnstable%']::text[]);

tract_name
----------------------------------------------------
Census Tract 102.06, Barnstable County, Massachusetts
Census Tract 103.06, Barnstable County, Massachusetts
Census Tract 106, Barnstable County, Massachusetts
Census Tract 9900, Dukes County, Massachusetts
(4 rows)
```

可以把这个例子里面的数组元素全部拆出来，每个写成一行 ILIKE 比较语句并用 OR 将这些语句连起来，这也可以达到与上例同等的效果，但显然要繁琐得多。也可以将 ANY 运算符与 LIKE、=、~（基于正则表达式的 like 运算）等比较运算符联用。

ANY 运算符是非常通用的，它可以针对任何数据类型的数组使用，可以与任何比较运算符联用（所谓比较运算符就是指返回结果类型为布尔型的运算符），甚至可以与用户自定义的或者是安装扩展包得到的新数据类型或者新比较运算符联用。

7.2.7 可以返回结果集的函数

所谓返回结果集的函数就是能够一次性返回多行记录的函数。

PostgreSQL 允许在 SELECT 语句中调用能够返回多行记录的函数。其他大多数数据库都不支持该功能，一般仅支持调用返回一行记录的函数。

在一个复杂的 SQL 语句中使用返回结果集的函数很容易导致意外的结果，这是因为这类函数输出的结果集会与该语句其他部分生成的结果集产生笛卡儿积，从而生成更多的记录行。因此在使用之前，你必须对此后果有所了解。在示例 7-14 中，我们使用 generate_series 函数演示了这种情况。先建表如下：

```
CREATE TABLE interval_periods (i_type interval);
INSERT  INTO  interval_periods (i_type)
VALUES ('5 months'), ('132 days'), ('4862 hours');
```

示例 7-14　在 SELECT 语句中使用返回结果集的函数

```
SELECT i_type,
    generate_series('2012-01-01'::date,'2012-12-31'::date,i_type) As dt
FROM interval_periods;

i_type     | dt
-----------+----------------------
5 months   | 2012-01-01 00:00:00-05
5 months   | 2012-06-01 00:00:00-04
5 months   | 2012-11-01 00:00:00-04
132 days   | 2012-01-01 00:00:00-05
132 days   | 2012-05-12 00:00:00-04
132 days   | 2012-09-21 00:00:00-04
4862 hours | 2012-01-01 00:00:00-05
4862 hours | 2012-07-21 15:00:00-04
```

7.2.8　限制对继承表的DELETE、UPDATE、INSERT操作的影响范围

如果表间是继承关系，那么查询父表时就会将子表中满足条件的记录也查出来。DELETE 和 UPDATE 操作也遵循类似逻辑，即对父表的修改操作也会影响到子表的记录。有时你可能希望操作仅限定于主表范围之内，并不希望子表受到波及。

PostgreSQL 提供了 ONLY 关键字以实现此功能。我们在示例 7-37 中展示了 ONLY 的用法。在该示例中，我们希望仅从生产表中删除那些尚未迁移到日志表中的记录。如果没有 ONLY 修饰符，我们最终将从子表中删除先前可能已移动过的记录。

7.2.9　DELETE USING语法

我们经常会遇到“只有当记录的字段值落在另外一个结果集中时，才需要删除该记录”的情况，那么此时就必须借助一次关联查询才能定位到要删除的目标记录。此时可以使用 USING 子句来指定需关联的表，然后在 WHERE 子句中编写 USING 子句的表和 FROM 子句的表之间的关联语句，来确定哪些记录需删除。USING 子句中可以指定多张表，中间用逗号分隔。在示例 7-15 中，我们借助一个关联查询实现了删除 census.facts 表中符合 short_name='s01' 这个条件的记录。

示例 7-15　DELETE USING 的用法

```
DELETE FROM census.facts
USING census.lu_fact_types As ft
WHERE facts.fact_type_id = ft.fact_type_id AND ft.short_name = 's01';
```

如果不使用该语法，一般的做法是在 WHERE 子句中使用笨拙的 IN 表达式。

7.2.10　将修改影响到的记录行返回给用户

RETURNING 是 ANSI SQL 规定的标准语法，但支持该语法的数据库却不多。在示例 7-37 中，我们通过 RETURNING 子句将在 DELETE 操作中被删除的记录返回给了用户。当然，INSERT 和

UPDATE 操作也是可以使用 RETURNING 的。对于带 serial 类型字段的表来说，RETURNING 语法是很有用的，因为向这类表中插入记录时，serial 字段是临时生成而非用户指定的；也就是说在插入动作完成之前，用户也不知道 serial 字段的值会是多少，除非是再查询一遍。而 RETURNING 语法使得用户不用再次查询就立即得到了 serial 字段的值。最常见的用法一般是 RETURNING *，即返回所有字段的值，但也可以指定仅返回特定字段，如示例 7-16 所示。

示例 7-16　在 UPDATE 语句中使用 RETURNING 子句返回修改过的记录

```
UPDATE census.lu_fact_types AS f
SET short_name = replace(replace(lower(f.fact_subcats[4]),' ','_'),':','')
WHERE f.fact_subcats[3] = 'Hispanic or Latino:' AND f.fact_subcats[4] > ''
RETURNING fact_type_id, short_name;

fact_type_id  | short_name
--------------+-----------------------------------------------------
96            | white_alone
97            | black_or_african_american_alone
98            | american_indian_and_alaska_native_alone
99            | asian_alone
100           | native_hawaiian_and_other_pacific_islander_alone
101           | some_other_race_alone
102           | two_or_more_races
```

7.2.11　UPSERT：INSERT时如果主键冲突则进行UPDATE

PostgreSQL 9.5 中引入了新的 INSERT ON CONFLICT 语法，业界一般也会将该功能称为 UPSERT。当无法确定即将插入的记录是否会与表中现有的记录冲突时，就可以使用 UPSERT 语法来确保不管是否有冲突都不会报错，该语法允许用户自定义发生冲突时的行为，要么更新现有记录，要么直接忽略。

使用该特性时，表上一定要有一个唯一性约束。这个约束可以是唯一性字段、主键、唯一索引或者排他性约束。一旦违反该约束，用户可以决定后续如何处理：用新记录覆盖现有记录，或者什么也不做。我们通过一个例子来演示其用法，假设我们有张颜色表：

```
CREATE TABLE colors(color varchar(50) PRIMARY KEY, hex varchar(6));
    INSERT INTO colors(color, hex)
    VALUES('blue', '0000FF'), ('red', 'FF0000');
```

上面刚刚插入了一条记录，接下来继续插入一些可能会与表中已有记录的主键重复的记录到该表中。正常情况下，插入主键重复的记录时会报主键冲突错误。在示例 7-17 中，通过使用 UPSERT 我们实现了对主键冲突错误的忽略，最后的结果是只有表中原来不存在的 green 记录被插入成功。如果后续再次执行该 SQL，由于所有主键都已存在，最终效果是不会往表中插入任何新记录。

示例 7-17　发生主键冲突时忽略冲突记录

```
INSERT INTO colors(color, hex)
   VALUES('blue', '0000FF'), ('red', 'FF0000'), ('green', '00FF00')
ON CONFLICT DO NOTHING ;
```

上述语句的保护力度还不够。如果有人插入了一条首字母大写的“Blue”的记录，由于与现有的“blue”并没有主键冲突，因此会插入成功，那么系统中实际上会出现两条同样表达“蓝色”的记录，这是我们不希望见到的。此时可以通过创建一个唯一索引来解决，语法如下所示。

```
CREATE UNIQUE INDEX uidx_colors_lcolor ON colors USING btree(lower(color));
```

此时再次插入“Blue”记录则会被唯一索引阻止，而且由于我们定义了 ON CONFLICT DO NOTHING，也就是冲突时什么都不干，所以效果相当于什么都没发生。但如果我们真的希望用“Blue”来取代表中已有的“blue”，则可以使用示例 7-18 的写法。

示例 7-18　ON CONFLIECT DO UPDATE 用法之一

```
INSERT INTO colors(color, hex)
  VALUES('Blue', '0000FF'), ('Red', 'FF0000'), ('Green', '00FF00')
ON CONFLICT(lower(color))
    DO UPDATE SET color = EXCLUDED.color, hex = EXCLUDED.hex;
```

在示例 7-18 中我们设定了 ON CONFLICT 子句后跟的唯一性字段为 lower(color)，该字段上需要定义有唯一约束或者唯一索引，因此如果把目标字段设定为 upper(color) 是没有意义的，因为 colors 表上没有基于 uppser(color) 的唯一索引。

当使用 ON CONFLICT DO UPDATE 这种搭配时，ON CONFLICT 后需要跟唯一性字段或者唯一约束名。如果后跟唯一性约束，则需要使用 ON CONFLICT ON CONSTRAINT + *约束名*的语法，如示例 7-19 所示。

示例 7-19　ON CONFLICT DO UPDATE 用法之二

```
INSERT INTO colors(color, hex)
        VALUES('Blue', '0000FF'), ('Red', 'FF0000'), ('Green', '00FF00')
ON CONFLICT ON CONSTRAINT colors_pkey
DO UPDATE SET color = EXCLUDED.color, hex = EXCLUDED.hex;
```

只有当违反主键约束、唯一索引或者唯一键值约束时，DO 子句才会被触发，整体语句不会报错。但如果发生了数据类型错误或是违反检查约束等别的错误，整体语句还是会报错，DO UPDATE 子句不会被触发。

7.2.12　在查询中使用复合数据类型

PostgreSQL 会在创建表时自动创建一个结构与表完全相同的数据类型，其中包含了多个其他数据类型的成员字段，因此也会被称为复合数据类型。你第一次见到基于复合数据类型的查询语句时，可能会感到很惊讶。事实上，你可能已经在编写调试 SQL 语句的过程中见识过它的神奇之处。先看一下这个语句：

```
SELECT x FROM census.lu_fact_types As x LIMIT 2;
```

第一眼看到这个语句时，你可能认为我们漏写了一个 .*，但请看一下该语句的执行结果：

```
x
------------------------------------------------------------------
(86,Population,"{D001,Total:}",d001)
(87,Population,"{D002,Total:,""Not Hispanic or Latino:""}",d002)
```

语句不但没有报错，而且返回的结果是标准的 lu_fact_type 类型。我们看一下第一行记录的内容来确认有没有问题，86 是 fact_type_id 字段的值，Population 是 category 字段的值，{D001,Total:} 是 fact_subcats 属性。可以看到，字段值与表定义完全匹配，没有问题。复合数据类型可以作为多个很有用的函数的输入，比如 array_agg 和 hstore（hstore 扩展包提供的一个函数，可以将一行记录转换为 hstore 的一个键值对象）等。

如果你正在开发 Web 应用，那么建议你充分利用 PostgreSQL 原生支持的 JSON 和 JSONB 数据类型的强大能力。关于 JSON 和 JSONB 的详情，请参见 5.6 节。通过联用 array_agg 和 array_to_json 这两个函数，可以将语句的查询结果转换为一个 JSON 对象后输出，如示例 7-20 所示。

示例 7-20　将查询结果转换为 JSON 格式

```
SELECT array_to_json(array_agg(f)) As cat ❶
FROM (
    SELECT MAX(fact_type_id) As max_type, category ❷
    FROM census.lu_fact_types
    GROUP BY category
) As f;
```

输出结果如下：

```
cats
----------------------------------------------------
[{"max_type":102,"category":"Population"},
{"max_type":153,"category":"Housing"}]
```

❶ 定义一个名为 f 的子查询，可以从中查出记录。

❷ 使用 array_agg 函数将子查询返回的结果集聚合为一个数组，然后将这个结果集数组通过 array_to_json 转变为一个 JSON 字段。

PostgreSQL 9.3 版提供了一个名为 json_agg 的函数，该函数的效果相当于上面示例中 array_to_json 和 array_agg 联用的效果，但执行速度更快，使用起来也更方便。在示例 7-21 中，我们使用 json_agg 改写了示例 7-20 中的语句，二者的输出是相同的。

示例 7-21　使用 json_agg 将查询结果转为 JSON 格式

```
SELECT json_agg(f) As cats
FROM (
    SELECT MAX(fact_type_id) As max_type, category
    FROM census.lu_fact_types
    GROUP BY category
) As f;
```

7.2.13　使用$文本引用符

在 ANSI SQL 标准中，字符串的引用符是单引号（'）。但如果字符串内容本身就含有单引号则会遇到麻烦，因为系统不知道字符串到底在哪里截止，此时就需要用转义符来对文本中的单引号进行转义。文本中含单引号的情况还是比较普遍的，比如名字 O’Nan，再比如从属关系表达 mon’s place，又比如缩写 can’t。我们需要采用在前面另加一个单引号的方式，对单引号进行转义。也就是说，字符串内容中看起来是两个单引号连写，但其实表达

的是“单个单引号”这一内容。比如你需要编写一个 INSERT 语句，其中某个字段的内容是从一部小说中复制出来的大段文字，其中难免会有单引号。如果在这大段文字中的每个单引号之前都附加一个单引号来进行转义，这项工作会很繁琐，而且文本读起来会让人很痛苦，因为两个单引号看起来很像是一个双引号，但二者又根本不是一回事。

为了解决这个问题，PostgreSQL 支持使用 $$ 来将任意长度的文本包起来，这样就可以不用对文本中的单引号做转义处理。

$ 引用符还可用于执行动态 SQL 的场景，例如 exec(某 sql)。在示例 7-5 中，我们就使用了 $$ 将触发器的定义字符串包起来。

如果需要用一个 SQL 来将两个字符串拼接起来，而这两个字符串中又含有很多单引号，那么符合 ANSI 标准的写法是这样的：

```
SELECT 'It''s O''Neil''s play. ' || 'It''ll start at two o''clock.'
```

使用 $$ 引用符看起来是这样的：

```
SELECT $$It's O'Neil's play. $$ || $$It'll start at two o'clock.$$
```

这显然简化了很多，也更加清晰易懂了。

前后的 $$ 符取代了原本的单引号文本引用符，并对其中所有的单引号实现了转义。

该用法还有一个变种，我们称之为**命名 $ 引用符**，后续内容中会对其进行介绍。

7.2.14 DO

DO 命令可以执行一个基于过程式语言的匿名代码段，可以将其看作一个一次性使用的匿名函数。在下面的示例中，我们将演示如何通过执行一个匿名代码段来将示例 3-10 中插入的数据从中间表加载到产品表中。示例中的匿名代码段是用 PL/pgSQL 编写的，但你也可以使用别的语言编写。

首先执行建表操作：

```
set search_path=census;
DROP TABLE IF EXISTS lu_fact_types CASCADE;
CREATE TABLE lu_fact_types (
    fact_type_id serial,
    category varchar(100),
    fact_subcats varchar(255)[],
    short_name varchar(50),
    CONSTRAINT pk_lu_fact_types PRIMARY KEY (fact_type_id)
);
```

然后使用 DO 语法来对其生成一些记录，如示例 7-22 所示。上面的 CASCADE 关键字的效果是将该表所有的关联对象一次性删除，比如外键约束、视图等。因此，使用 CASCADE 关键字时要特别小心。

示例 7-22 中生成了一系列的 INSERT INTO SELECT 语句，然后通过这些语句实现数据迁移。这些 SQL 还实现了由列转行的转换操作。



示例 7-22 中仅包含了为 lu_fact_types 表生成数据的部分代码。请从本书附加的代码和数据资源包中找到 building_census_tables.sql 这个脚本，其中有完整的建表语句。

示例 7-22　使用 DO 命令来生成动态 SQL

```
DO language plpgsql
$$
DECLARE var_sql text;
BEGIN
    var_sql := string_agg(
        $sql$ ❶
        INSERT INTO lu_fact_types(category, fact_subcats, short_name)
        SELECT
            'Housing',
            array_agg(s$sql$ || lpad(i::text,2,'0')
              || ') As fact_subcats,'
              || quote_literal('s' || lpad(i::text,2,'0')) || ' As short_name
        FROM staging.factfinder_import
        WHERE s' || lpad(I::text,2,'0') || $sql$ ~ '^[a-zA-Z]+' $sql$, ';'
    )
    FROM generate_series(1,51) As I; ❷
    EXECUTE var_sql; ❸
END
$$;
```

❶ 这里使用了 $ 引用符，因此不需要为后面的 Housing 前后的单引号进行转义。因为 DO 之后的命令是用 $$ 引用符封装起来的，所以这里需要使用命名的 $ 引用符而不能使用 $$。我们选择的是 sql 这个命名引用符。

❷ 使用 string_agg 函数让一组 SQL 语句形成单一字符串的形式 INSERT INTO lu_fact_type(...) SELECT ... WHERE s01 ~ '[a-zA-Z]+';。

❸ 执行该 SQL。

在示例 7-22 中，使用了 7.2.13 节中介绍过的 $ 引用符，来将 DO 之后的函数体及函数体内部的一些 SQL 语句包裹了起来。由于同时使用了内外两级 $ 引用符，所以如果对这两级引用都使用默认的 $$ 引用符，会造成系统无法分清每一级引用符的边界在哪里。此时我们可以对内外两级 $ 引用符中的至少一级命名（也就是所谓的“命名 $ 引用符”）以示区分，当然，把内外两级都改为命名引用符也是可以的。

7.2.15　适用于聚合操作的FILTER子句

9.4 版中新引入了用于聚合操作的 FILTER 子句，这是近期 ANSI SQL 标准中新加入的一个关键字。该关键字用于替代同为 ANSI SQL 标准语法的 CASE WHEN 子句，使聚合操作的语法得以简化。例如，假设你需要使用 CASE WHEN 子句来统计每个学生不同科目的多次测试的平均成绩，语法如示例 7-23 所示。

示例 7-23　在 AVG 聚合函数中使用 CASE WHEN

```
SELECT student,
    AVG(CASE WHEN subject ='algebra' THEN score ELSE NULL END) As algebra,
```

```
    AVG(CASE WHEN subject ='physics' THEN score ELSE NULL END) As physics
FROM test_scores
GROUP BY student;
```

用 FILTER 子句可以实现与上面语句等价的效果，语法如的示例 7-24 所示。

示例 7-24　AVG 聚合函数与 FILTER 子句的配合使用

```
SELECT student,
    AVG(score) FILTER (WHERE subject ='algebra') As algebra,
    AVG(score) FILTER (WHERE subject ='physics') As physics
FROM test_scores
GROUP BY student;
```

对于求平均值、求合计值以及其他很多聚合函数来说，CASE 和 FILTER 子句是等价的，即二者可以起到相同的作用。FILTER 子句的优势在于写法比较清晰简洁，并且操作大数据量时速度比较快。CASE 语句对于筛选掉的字段值是当成 NULL 处理的，因此对于 array_agg 这种会处理 NULL 值的聚合函数来说，使用 CASE WHEN 子句就不止是写法繁琐的问题了，还会导致输出不想要的结果。在示例 7-25 中，我们使用 CASE...WHEN... 方法查询每个学生的各门课程的多次测试成绩的列表来演示这个问题。

示例 7-25　CASE WHEN 子句与 array_agg 函数配合使用

```
SELECT student,
    array_agg(CASE WHEN subject ='algebra' THEN score ELSE NULL END) As algebra,
    array_agg(CASE WHEN subject ='physics' THEN score ELSE NULL END) As physics
FROM test_scores
GROUP BY student;

student | algebra                    | physics
--------+---------------------------+------------------------------
jojo    | {74,NULL,NULL,NULL,74,..} | {NULL,83,NULL,NULL,NULL,79,..}
jdoe    | {75,NULL,NULL,NULL,78,..} | {NULL,72,NULL,NULL,NULL,72..}
robe    | {68,NULL,NULL,NULL,77,..} | {NULL,83,NULL,NULL,NULL,85,..}
lhsu    | {84,NULL,NULL,NULL,80,..} | {NULL,72,NULL,NULL,NULL,72,..}
(4 rows)
```

可以看到示例 7-25 输出的成绩列表中含有很多的 NULL 值。这个问题可以通过使用子查询来解决，但比起使用 FILTER 来说还是麻烦又低效。示例 7-26 中演示了使用 FILTER 时的写法。

示例 7-26　FILTER 子句与 array_agg 函数的配合使用

```
SELECT student,
    array_agg(score) FILTER (WHERE subject ='algebra') As algebra,
    array_agg(score) FILTER (WHERE subject ='physics') As physics
FROM test_scores
GROUP BY student;

student | algebra | physics
--------+---------+--------
jojo    | {74,74} | {83,79}
jdoe    | {75,78} | {72,72}
robe    | {68,77} | {83,85}
lhsu    | {84,80} | {72,72}
```

FILTER 子句适用于所有聚合函数，不仅仅是 PostgreSQL 中内置的那些聚合函数，通过安装扩展包支持的聚合函数也是可以用的。

7.2.16 查询百分位数与最高出现频率数

PostgreSQL 9.4 中开始支持用于计算百分位数、中位数（等同于 0.5 百分位数）和最高出现频率数的统计函数，具体包括 percentile_disc（用于计算离散百分位）、percentile_cont（用于计算连续百分位）以及 mode 这三个函数。

percentile_disc 和 percentile_cont 的区别在于二者处理同一百分位命中多条记录的方法。前者会取该百分位范围内的第一条命中记录的值，因此记录顺序会对最终返回哪条记录产生影响；后者会把百分位范围内命中的所有记录取均值后返回。

中位数就是百分位值等于 0.5 的百分位数，因此计算中位数不需要一个专门的函数。mode 函数的作用是取分组中出现频率最高的值，如果最高频率的值有多个，则取排序后的第一个，因此排序方法会对 mode 计算的结果有影响，如示例 7-27 所示。

示例 7-27　计算中位数和出现频率最高的成绩

```
SELECT
    student,
    percentile_cont(0.5) WITHIN GROUP (ORDER BY score) As cont_median,
    percentile_disc(0.5) WITHIN GROUP (ORDER BY score) AS disc_median,
    mode() WITHIN GROUP (ORDER BY score) AS mode,
    COUNT(*) As num_scores
FROM test_scores
GROUP BY student
ORDER BY student;

student | cont_median | disc_median | mode | num_scores
--------+-------------+-------------+------+------------
alex    |          78 |          77 |   74 |          8
leo     |          72 |          72 |   72 |          8
regina  |          76 |          76 |   68 |          9
sonia   |        73.5 |          72 |   72 |          8
(4 rows)
```

示例 7-27 同时以离散模式和连续模式计算了学生成绩的中位数，当出现同分的学生时，计算结果可能是不一样的。

一般的聚合函数是直接把要进行聚合运算的目标字段作为入参，但前面介绍的这几个函数不同，它们的直接入参是百分位数或者为空，目标聚合字段是通过后面的 WITHIN GROUP 子句中的 ORDER BY 修饰符来指定的。

前述两个中位数计算函数还有一种用法，即可以输入多个数字作为百分位数，从而实现一次查询返回匹配多个百分位数的多个目标记录。示例 7-28 通过单条 SQL 语句实现了一次性获取中位数记录、60% 分位数记录以及最高分记录。

示例 7-28　一次性计算多个百分位数

```
SELECT
    student,
```

```
    percentile_cont('{0.5,0.60,1}'::float[])
WITHIN GROUP (ORDER BY score) AS cont_median,
    percentile_disc('{0.5,0.60,1}'::float[])
WITHIN GROUP (ORDER BY score) AS disc_median,
COUNT(*) As num_scores
FROM test_scores
GROUP BY student
ORDER BY student;

student | cont_median     | disc_median | num_scores
--------+-----------------+-------------+-----------
alex    | {78,79.2,84}    | {77,79,84}  | 8
leo     | {72,73.6,84}    | {72,72,84}  | 8
regina  | {76,76.8,90}    | {76,77,90}  | 9
sonia   | {73.5,75.6,86}  | {72,75,86}  | 8
(4 rows)
```

如同其他聚合函数一样，你可以将这几个函数与聚合函数中一些通用的修饰符联用。示例 7-29 演示了如何将 `WITHIN GROUP` 与 `FILTER` 联用。

示例 7-29　计算两个科目的成绩中位数

```
SELECT
    student,
    percentile_disc(0.5) WITHIN GROUP (ORDER BY score)
        FILTER (WHERE subject = 'algebra') AS algebra,
    percentile_disc(0.5) WITHIN GROUP (ORDER BY score)
        FILTER (WHERE subject = 'physics') AS physics
FROM test_scores
GROUP BY student
ORDER BY student;

student | algebra | physics
--------+---------+--------
alex    | 74      | 79
leo     | 80      | 72
regina  | 68      | 83
sonia   | 75      | 72
(4 rows)
```

7.3　窗口函数

窗口函数是 ANSI SQL 标准中规定的一个通用特性。通过使用窗口函数，可以在当前记录行中访问到与其存在特定关系的其他记录行，相当于在每行记录上都开了一个访问外部数据的窗口，这也是“窗口函数”这个名称的由来。“窗口”就是当前行可见的外部记录行的范围。通过窗口函数可以把当前行的“窗口”区域内的记录的聚合运算结果附加到当前记录行。`row_number` 和 `rank` 这类窗口函数能够基于窗口区的数据实现对记录行的复杂排序。

如果不借助窗口函数而又想要达到相同的效果，就只能使用关联操作和子查询。表面上看，使用窗口函数违背了 SQL 语言“基于结果集”的编程思想，因为它为每一行数据拓展出了一个外部数据域。但从另外一个角度看，我们可以认为窗口函数本质上仅是一种用

来替代关联操作和子查询的简写语法，也就是说窗口函数并未突破 SQL 体系原有的运算逻辑，那么也就不算违反了“基于结果集”的思想。你可以从 PostgreSQL 官方手册的“窗口函数”一节中看到更多说明和示例。

示例 7-30 可帮助你理解窗口函数的基本概念。通过使用窗口函数，可以在单个 SELECT 语句中同时获取到符合 fact_type_id=86 条件的记录的均值计算结果，以及原始记录的详细信息。请注意，语句执行时总是先筛选 WHERE 条件再计算窗口函数，因为这样显然可以避免做无用功。

示例 7-30　基本的窗口函数

```
SELECT tract_id, val, AVG(val) OVER () as val_avg
FROM census.facts
WHERE fact_type_id = 86;

tract_id    | val      | val_avg
------------+----------+----------------------
25001010100 | 2942.000 | 4430.0602165087956698
25001010206 | 2750.000 | 4430.0602165087956698
25001010208 | 2003.000 | 4430.0602165087956698
25001010304 | 2421.000 | 4430.0602165087956698
:
```

OVER 子句限定了窗口中的可见记录范围。本例中的 OVER 子句未设定任何条件，因此从该窗口中能看见全表的所有记录，所以 AVERAGE 运算的结果就是表中所有符合 fact_type_id=86 条件的记录中 val 字段的平均值。你可以看到，通过为其增加 OVER 子句，我们把一个传统的 AVG 聚合运算函数转变成了一个窗口函数。PostgreSQL 在遍历每一行记录时都会基于全表记录进行一次 AVG 运算，然后将得到的均值作为当前行的一个字段输出。由于窗口数据域内包含多条记录，这意味着窗口函数运算的结果一定在多条记录上都是重复的。事实上，窗口函数实现了无须 GROUP BY 的聚合运算，还实现了无须 JOIN 的关联操作，从而将窗口函数的运算结果回填到记录行中。

所有 SQL 聚合函数都可以通过增加 OVER 子句的方式来当作窗口函数使用。除了这些双重身份的函数之外，系统中还有 ROW、RANK、LEAD 等专门的窗口函数，你可以从 PostgreSQL 官方手册的“窗口函数”一节中看到完整的窗口函数列表。

7.3.1　PARTITION BY子句

窗口函数的窗口可见记录范围是可设置的，可以是全表记录，也可以是与当前行有关联关系的特定记录行。窗口可见记录范围的设置是通过 PARTITION BY 子句实现的，它可以指示 PostgreSQL 仅在满足条件的特定记录集上执行聚合操作。示例 7-31 的查询与示例 7-30 类似，但要求各县级编号作为窗口筛选条件，该编号就是 tract_id 的前 5 个字符。这样就实现了每个县都计算各自的平均值。

示例 7-31　使用县级编号作为窗口可见记录范围的筛选条件

```
SELECT tract_id, val, AVG(val) OVER (PARTITION BY left(tract_id,5)) As val_avg_county
FROM census.facts
WHERE fact_type_id = 2 ORDER BY tract_id;
```

```
tract_id    | val      | val_avg_county
------------+----------+---------------------
25001010100 | 1765.000 | 1709.9107142857142857
25001010206 | 1366.000 | 1709.9107142857142857
25001010208 |  984.000 | 1709.9107142857142857
:
25003900100 | 1920.000 | 1438.2307692307692308
25003900200 | 1968.000 | 1438.2307692307692308
25003900300 | 1211.000 | 1438.2307692307692308
:
```

7.3.2 ORDER BY子句

窗口函数的 OVER 子句中还可以使用 ORDER BY 子句，其作用可以理解为对窗口可见范围内的所有记录进行排序，并且窗口可见记录域是从结果集的第一条记录开始到当前记录为止的范围内。该语法的典型应用场景就是用 ROW_NUMBER 函数对记录集编号。示例 7-32 演示了如何对各人口普查区记录按照其名称顺序进行编号。

示例 7-32　使用 ROW_NUMBER 窗口函数进行编号操作

```
SELECT ROW_NUMBER() OVER (ORDER BY tract_name) As rnum, tract_name
FROM census.lu_tracts
ORDER BY rnum LIMIT 4;

rnum | tract_name
-----+-------------------------------------------------
1    | Census Tract 1, Suffolk County, Massachusetts
2    | Census Tract 1001, Suffolk County, Massachusetts
3    | Census Tract 1002, Suffolk County, Massachusetts
4    | Census Tract 1003, Suffolk County, Massachusetts
```

示例 7-32 中有两个 ORDER BY，前一个在 OVER 子句内生效，表明窗口可见区内的记录顺序，后一个针对整句生效，表明返回记录的整体顺序。请不要将二者的作用域混淆。

PARTITION BY 和 ORDER BY 可以联用，其效果就是对 PARTITION BY 指定的记录集进行排序。示例 7-33 还是复用了前面的例子，但在 OVER 子句中联用了 PARTITION BY 和 ORDER BY。

示例 7-33　联用 PARTITION BY 和 ORDER BY

```
SELECT tract_id, val,
    SUM(val) OVER (PARTITION BY left(tract_id,5) ORDER BY val) As sum_county_ordered
FROM census.facts
WHERE fact_type_id = 2
ORDER BY left(tract_id,5), val;

tract_id    | val      | sum_county_ordered
------------+----------+-----------------
25001014100 | 226.000 |          226.000
25001011700 | 971.000 |         1197.000
25001010208 | 984.000 |         2181.000
:
25003933200 | 564.000 |          564.000
25003934200 | 593.000 |         1157.000
25003931300 | 606.000 |         1763.000
:
```

可以看到，上面输出的合计值是逐行累加的，这就是在 OVER 子句中应用了 ORDER BY 后的效果，即窗口可见域是从排序后的记录集的头条记录开始，到 ORDER BY 字段值与当前记录值匹配的那行记录为止，因此最终会呈现为动态累加的效果。例如，对于第三个数据分区中的第五条记录来说，合计值仅会包含该分区中的前五条记录的值。在上面的示例中，我们在语句的最后加上了 ORDER BY left(tract_id,5), val 这个排序动作，因此动态累加效果一目了然。但请一定要牢记，OVER 子句中的 ORDER BY 与整句尾部的 ORDER BY 的作用是完全不同的。

你还可以通过 RANGE 或者 ROWS 关键字来显式指定窗口的可见记录域。例如：ROWS BETWEEN CURRENT ROW AND 5 FOLLOWING。

PostgreSQL 还支持建立命名窗口，该功能适用于在同一个查询中使用了多个窗口函数，且每个窗口函数的窗口定义都相同的情况。示例 7-34 演示了建立命名窗口的方法，同时还展示了 LEAD 和 LAG 窗口函数的用法，这两个窗口函数可以取出当前窗口中排在当前记录行之前或者之后的记录。

示例 7-34 命名窗口以及 LEAD 和 LAG 函数的用法

```
SELECT * FROM (
    SELECT
        ROW_NUMBER() OVER( wt ) As rnum, ❶
        substring(tract_id,1, 5) As county_code,
        tract_id,
        LAG(tract_id,2) OVER wt As tract_2_before,
        LEAD(tract_id) OVER wt As tract_after
    FROM census.lu_tracts
    WINDOW wt AS (PARTITION BY substring(tract_id,1, 5) ORDER BY tract_id) ❷
) As x
WHERE rnum BETWEEN 2 and 3 AND county_code IN ('25007','25025')
ORDER BY county_code, rnum;

rnum | county_code | tract_id    | tract_2_before | tract_after
-----+-------------+-------------+----------------+------------
2    | 25007       | 25007200200 |                | 25007200300
3    | 25007       | 25007200300 | 25007200100    | 25007200400
2    | 25025       | 25025000201 |                | 25025000202
3    | 25025       | 25025000202 | 25025000100    | 25025000301
```

❶ 直接复用窗口名，而不需要把窗口的完整定义再输一遍。

❷ 将我们的窗口命名为 wt 窗口。

LEAD 和 LAG 函数都有一个可选的 step 参数，该参数可以是正数也可以是负数，代表需要从当前记录开始向前或者向后跳几条记录才能访问到目标记录。当 LEAD 和 LAG 在寻找目标记录的过程中跳出了当前窗口的可见域时，就会返回 NULL。这种情况经常会遇到。

请注意：在 PostgreSQL 中，系统自带的以及用户自定义的聚合函数都可以作为窗口函数使用，但其他数据库一般仅支持 AVG、SUM、MIN、MAX 这些系统内置人聚合函数作为窗口函数使用。

7.4 CTE表达式

公用表表达式（CTE）本质上来说就是在一个非常庞大的 SQL 语句中，允许用户通过一个子查询语句先定义出一个临时表，然后在这个庞大的 SQL 语句的不同地方都可以直接使用这个临时表。CTE 本质上就是当前语句执行期间内有效的临时表，一旦当前语句执行完毕，其内部的 CTE 表也随之失效。

有以下三种类型的 CTE。

基本 CTE

这是最普通的 CTE，它可以使 SQL 语句的可读性更高，同时规划器在解析到这种 CTE 时会判定其查询代价是否很高，如果是的话，会考虑将其查询结果临时物化存储下来（此处概念跟物化视图非常类似），这样整个 SQL 语句的其他部分再访问此 CTE 时就会更快。

可写 CTE

这是对基本 CTE 的一个功能扩展，其内部可以执行 `UPDATE`、`INSERT` 或者 `DELETE` 操作。该类 CTE 最后一般会返回修改后的记录集。

递归 CTE

该类 CTE 在普通 CTE 的基础上增加了一个循环操作。在执行过程中，递归 CTE 返回的结果集会有所变化。

PostgreSQL 支持可写的递归 CTE 这种复合类型。

7.4.1 基本CTE用法介绍

基本 CTE 的用法如示例 7-35 所示。`WITH` 关键字后面跟着的就是 CTE 表达式。

示例 7-35　基本 CTE

```
WITH cte AS (
    SELECT
        tract_id, substring(tract_id,1, 5) As county_code,
        COUNT(*) OVER(PARTITION BY substring(tract_id,1, 5)) As cnt_tracts
    FROM census.lu_tracts
)
SELECT MAX(tract_id) As last_tract, county_code, cnt_tracts
FROM cte
WHERE cnt_tracts > 100
GROUP BY county_code, cnt_tracts;
```

示例 7-35 中 CTE 表达式的名称是 `cte`，其本体是由一个 `SELECT` 语句定义出来的，查询字段列表中包含 `tract_id`、`country_code`、`cnt_tracts` 三个列。外围的 SQL 语句会将该 CTE 作为一个临时表来使用。

单个 SQL 语句中可以创建多个 CTE，CTE 之间使用逗号分隔，所有的 CTE 表达式都要落在 `WITH` 子句范围内，具体语法如示例 7-36 所示。多个 CTE 表达式之间的顺序不是随便排的，排在后面的 CTE 可以引用排在前面的 CTE，但反过来不行。除了这一点以外，多个

CTE 之间的排列顺序没有别的讲究。

示例 7-36　多个 CTE 的用法

```
WITH
    cte1 AS(
        SELECT
            tract_id,
            substring(tract_id,1, 5) As county_code,
            COUNT(*) OVER (PARTITION BY substring(tract_id,1,5)) As cnt_tracts
        FROM census.lu_tracts
    ),
    cte2 AS (
        SELECT
            MAX(tract_id) As last_tract,
            county_code,
            cnt_tracts
        FROM cte1
        WHERE cnt_tracts < 8 GROUP BY county_code, cnt_tracts
    )
SELECT c.last_tract, f.fact_type_id, f.val
FROM census.facts As f INNER JOIN cte2 c ON f.tract_id = c.last_tract;
```

7.4.2　可写CTE用法介绍

可写 CTE 扩展了 CTE 的功能范畴，从只读扩展为可写。下面使用示例 6-3 中创建的日志表来演示此功能。首先创建一个子表：

```
CREATE TABLE logs_2011_01_02 (
    PRIMARY KEY (log_id),
    CONSTRAINT chk
        CHECK (log_ts >= '2011-01-01' AND log_ts < '2011-03-01')
)
INHERITS (logs_2011);
```

在示例 7-37 中，我们将父表的部分数据迁移到子表中。父表包含了 2011 年全年的数据，子表包含了 2011 年 1 月和 2 月的数据。下面的语句中使用的 `ONLY` 关键字在 7.2.8 节中有相关介绍，而 `RETURNING` 关键字在 7.2.10 节中有相关介绍。

示例 7-37　使用可写 CTE 将数据从一个分支移动到另一个分支

```
WITH t AS (
    DELETE FROM ONLY logs_2011 WHERE log_ts < '2011-03-01' RETURNING *
)
INSERT INTO logs_2011_01_02 SELECT * FROM t;
```

7.4.3　递归CTE用法介绍

PostgreSQL 官方手册中对递归 CTE 做了最好的说明：“通过新增一个可选的 `RECURSIVE` 修饰符，使 CTE 从仅仅能提供一些语法便利升华为能够实现标准 SQL 语法无法实现的功能。”递归 CTE 能够使用递归语法构造出一个表达式，这一点很有意思。递归 CTE 使用 `UNION ALL` 语法来实现每次运算过程中的运算结果的递归累积。

如果要将一个基本 CTE 转换为递归 CTE，需要在 WITH 后加上 RECURSIVE 修饰符。WITH RECURSIVE 后面可以附带由递归表达式和非递归表达式结合而成的语句。在大多数其他数据库中，如果要表达递归关系，并不需要显式指定 RECURSIVE 关键字。

递归 CTE 常用于表达消息线程和其他树状结构。博文“Recursive CTE to Display Tree Structures”中提供了一个这方面的例子。

在示例 7-38 中，我们通过查询系统 catalog 来展示数据库中的级联表关系。

示例 7-38　递归 CTE

```
WITH RECURSIVE tbls AS (
    SELECT
        c.oid As tableoid,
        n.nspname AS schemaname,
        c.relname AS tablename ❶
    FROM
        pg_class c LEFT JOIN
        pg_namespace n ON n.oid = c.relnamespace LEFT JOIN
        pg_tablespace t ON t.oid = c.reltablespace LEFT JOIN
        pg_inherits As th ON th.inhrelid = c.oid
    WHERE
        th.inhrelid IS NULL AND
        c.relkind = 'r'::"char" AND c.relhassubclass
    UNION ALL
    SELECT
        c.oid As tableoid,
        n.nspname AS schemaname,
        tbls.tablename || '->' || c.relname AS tablename ❷ ❸
    FROM
        tbls INNER JOIN
        pg_inherits As th ON th.inhparent = tbls.tableoid INNER JOIN
        pg_class c ON th.inhrelid = c.oid LEFT JOIN
        pg_namespace n ON n.oid = c.relnamespace LEFT JOIN
    pg_tablespace t ON  t.oid = c.reltablespace
)
SELECT * FROM tbls ORDER BY tablename; ❹

tableoid | schemaname | tablename
---------+------------+-------------------------------------
3152249  | public     | logs
3152260  | public     | logs->logs_2011
3152272  | public     | logs->logs_2011->logs_2011_01_02
```

❶ 查询出所有有子表而无父表的表。

❷ 这是递归查询部分，查询出了所有位于 tbls 临时表中的表的子表。

❸ 输出时在父表名之后附加子表的名称。

❹ 输出父表和所有子表。因为语句中要求输出结果按照表名排序，而每一层级的表名是将父表的名称排在子表之前，所以排序后的效果就是子表记录紧跟在其父表记录之后输出。

7.5 LATERAL横向关联语法

LATERAL 是 9.3 版中新支持的 ANSI SQL 标准语法。该语法的用途是：假设你需要对两张表或者两个子查询进行关联查询操作，那么参与关联运算的双方是独立的，互相不能读取对方的数据。例如，下面的查询语句会报错，因为 L.year=2011 不是位于关联的右侧的一个列。

```
SELECT *
    FROM
        census.facts L
        INNER JOIN
        (
            SELECT *
            FROM census.lu_fact_types
            WHERE category = CASE WHEN L.yr = 2011
THEN 'Housing' ELSE category END
        ) R
    ON L.fact_type_id = R.fact_type_id;
```

加上了 LATERAL 关键字后就不会再报错：

```
SELECT *
    FROM
        census.facts L INNER JOIN LATERAL
        (
            SELECT *
            FROM census.lu_fact_types
            WHERE category = CASE WHEN L.yr = 2011
THEN 'Housing' ELSE category END
        ) R
        ON L.fact_type_id = R.fact_type_id;
```

通过使用 LATERAL 语法，可以在一个 FROM 子句中跨两个表共享多列中的数据。但有个限制就是仅支持单向共享，即右侧的表可以提取左侧表中的数据，但反过来不行。

有时候，为了避免编写语法极其复杂的语句，也需要使用 LATERAL 语法。在示例 7-39 中，关联关系中左侧的一个列充当了右侧 generate_series 函数的一个形参。

```
CREATE TABLE interval_periods(i_type interval);
INSERT INTO interval_periods (i_type)
VALUES ('5 months'), ('132 days'), ('4862 hours');
```

示例 7-39　LATERAL 语法和 generate_series 函数的关联使用

```
SELECT i_type, dt
FROM
    interval_periods CROSS JOIN LATERAL
    generate_series('2012-01-01'::date, '2012-12-31'::date, i_type) AS dt
WHERE NOT (dt = '2012-01-01' AND i_type = '132 days'::interval);

i_type      | dt
------------+----------------------
5 mons      | 2012-01-01 00:00:00-05
5 mons      | 2012-06-01 00:00:00-04
```

```
5 mons      | 2012-11-01 00:00:00-04
132 days    | 2012-05-12 00:00:00-04
132 days    | 2012-09-21 00:00:00-04
4862:00:00  | 2012-01-01 00:00:00-05
4862:00:00  | 2012-07-21 15:00:00-04
```

LATERAL 语法还可用于以下场景：通过关联关系左侧的数据来限制右侧的查询结果集中包含的记录数量。在示例 7-40 中，我们使用 LATERAL 语法查询出最近 100 天之内登录过我们网站（http://www.postgresonline.com）的超级用户最近 5 次登录的时间和操作日志。本例中使用的表是在 6.1.5 节和 6.1.1 节中创建的。

示例 7-40　使用 LATERAL 语法来限制关联查询中的一方返回的记录数

```
SELECT u.user_name, l.description, l.log_ts
FROM
    super_users AS u CROSS JOIN LATERAL (
    SELECT description, log_ts
    FROM logs
    WHERE
        log_ts > CURRENT_TIMESTAMP - interval '100 days' AND
        logs.user_name = u.user_name
    ORDER BY log_ts DESC LIMIT 5
    ) AS l;
```

虽然你也可以通过窗口函数来实现相同的效果，但 LATERAL 关联执行速度更快，语法也更简洁。

在同一条 SQL 语句中可以多次使用 LATERAL 关联，当需要关联多个子查询时，甚至可以级联使用 LATERAL。在有的场景下可以省略 LATERAL 关键字，此时规划器会根据关联关系的两边交叉引用的情况智能判断出这是一个 LATERAL 操作。但为了清晰起见，最好显式指定 LATERAL 关键字。在不支持 LATERAL 语法的 PostgreSQL 版本上执行带横向引用的 SQL 语句当然会报错。不显式指明 LATERAL 关键字还有一个风险，就是可能会造成规划器误判，最终生成的执行计划可能完全不是你想要的。

其他数据库中也提供了横向关联的能力，但其语法不符合 ANSI SQL 规范的要求。在 Oracle 中，横向关联通过管道函数实现，在 SQL Server 中使用 CROSS APPLY 或者 OUTER APPLY 语法来实现。

7.6　WITH ORDINALITY子句

PostgreSQL 9.4 开始支持 ANSI SQL 标准中规定的 WITH ORDINALITY 语法。WITH ORDINALITY 子句的作用是为函数返回的结果集中的每一行自动附加一个序列号字段。



在普通的表查询语句和子查询语句中不可以使用 WITH ORDINALITY 语法，但可以使用 ROW_NUMBER 窗口函数，效果相同。

你会发现，WITH ORDINALITY 经常与 generate_series、unnest 之类可以将复合数据类型和数组展开的函数联用。事实上，WITH ORDINALTY 语法可与任何返回多条记录的函数联用，

包括用户自定义的函数。

示例 7-41 演示了 WITH ORDINALITY 与 generate_series 函数生成的临时变量联用的场景。

示例 7-41　对函数返回的多条记录结果进行编号

```
SELECT dt.*
FROM generate_series('2016-01-01'::date,'2016-12-31'::date,interval '1 month')
WITH ORDINALITY As dt;

dt                     | ordinality
-----------------------+-----------
2016-01-01 00:00:00-05 |          1
2016-02-01 00:00:00-05 |          2
2016-03-01 00:00:00-05 |          3
2016-04-01 00:00:00-04 |          4
2016-05-01 00:00:00-04 |          5
2016-06-01 00:00:00-04 |          6
2016-07-01 00:00:00-04 |          7
2016-08-01 00:00:00-04 |          8
2016-09-01 00:00:00-04 |          9
2016-10-01 00:00:00-04 |         10
2016-11-01 00:00:00-04 |         11
2016-12-01 00:00:00-05 |         12
(12 rows)
```

WITH ORDINALITY 会在查询结果的最后增加一个名为 ordinality 的字段，WITH ORDINALITY 子句只能出现在 SQL 语句的 FROM 子句中。ordinality 字段可以改为其他名字。

WITH ORDINALITY 子句与 LATERAL 语法经常会联用。在示例 7-42 中，我们复用了示例 7-39 中的 LATERAL 例句，同时为返回的记录增加了一个序列号。

示例 7-42　将 WITH ORDINALITY 与 LATERAL 联用

```
SELECT d.ord, i_type, d.dt
FROM
    interval_periods CROSS JOIN LATERAL
    generate_series('2012-01-01'::date, '2012-12-31'::date, i_type)
WITH ORDINALITY AS d(dt,ord)
WHERE NOT (dt = '2012-01-01' AND i_type = '132 days'::interval);

ord | i_type     | dt
----+------------+-----------------------
  1 | 5 mons     | 2012-01-01 00:00:00-05
  2 | 5 mons     | 2012-06-01 00:00:00-04
  3 | 5 mons     | 2012-11-01 00:00:00-04
  2 | 132 days   | 2012-05-12 00:00:00-04
  3 | 132 days   | 2012-09-21 00:00:00-04
  1 | 4862:00:00 | 2012-01-01 00:00:00-05
  2 | 4862:00:00 | 2012-07-21 15:00:00-04
(7 rows)
```

在示例 7-42 中，WITH ORDINALITY 与返回多行记录的函数联用，它只能在 WHERE 子句之前的 FROM 部分生效，因此查询得到的最终结果中会出现序号非连续的情况（比如上例中的 132 days 对应的序号 1 就没有了），其原因是有的记录被 WHERE 条件过滤掉了。

7.7 GROUPING SETS、CUBE和ROLLUP语法

如果你需要创建一个包含总计和分子类总计的报表，GROUPING SETS 就是为这种场景量身定做的。

还是以前面使用的学生测试成绩表为例。如果需要同时统计每个学生的综合平均分和每个科目的平均分，可以写一个类似示例 7-43 的 SQL 语句来达成目标，其中使用了 GROUPING SETS 语法。

示例 7-43　统计每个学生的综合平均分以及每个学生分科目的平均分

```
SELECT student, subject, AVG(score)::numeric(10,2)
FROM test_scores
WHERE student IN ('leo','regina')
GROUP BY GROUPING SETS ((student),(student,subject))
ORDER BY student, subject NULLS LAST;

 student |  subject  | avg
---------+-----------+-------
 leo     | algebra   | 82.00
 leo     | calculus  | 65.50
 leo     | chemistry | 75.50
 leo     | physics   | 72.00
 leo     | NULL      | 73.75
 regina  | algebra   | 72.50
 regina  | calculus  | 64.50
 regina  | chemistry | 73.50
 regina  | economics | 90.00
 regina  | physics   | 84.00
 regina  | NULL      | 75.44
(11 rows)
```

示例 7-43 通过单个 SQL 语句就统计出了每个学生所有科目的平均分以及每个学生每个科目的平均分。

同理，我们甚至能够再加上每门科目所有学生的平均分，依然是借助 GROUPING SETS 的能力来实现，如示例 7-44 所示。

示例 7-44　统计每个学生的综合平均分、每个学生分科目的平均分以及某个科目所有学生的平均分

```
SELECT student, subject, AVG(score)::numeric(10,2)
FROM test_scores
WHERE student IN ('leo','regina')
GROUP BY GROUPING SETS ((student,subject),(student),(subject))
ORDER BY student NULLS LAST, subject NULLS LAST;

 student |  subject  | avg
---------+-----------+-------
 leo     | algebra   | 82.00
 leo     | calculus  | 65.50
 leo     | chemistry | 75.50
 leo     | physics   | 72.00
 leo     | NULL      | 73.75
```

```
 regina  | algebra   | 72.50
 regina  | calculus  | 64.50
 regina  | chemistry | 73.50
 regina  | economics | 90.00
 regina  | physics   | 84.00
 regina  | NULL      | 75.44
 NULL    | algebra   | 77.25
 NULL    | calculus  | 65.00
 NULL    | chemistry | 74.50
 NULL    | economics | 90.00
 NULL    | physics   | 78.00
(16 rows)
```

如果希望一次性统计出每个学生的综合平均分、每个学生分科目的综合平均分和全局综合平均分，该怎么办呢？可以把查询修改为 GROUPING SETS((student),(student, subject),())，这种递进式的分组聚合可以用 ROLLUP(student,subject) 来表达，具体参见示例 7-45。

示例 7-45　统计每个学生的综合平均分、每个学生分科目的平均分以及全局综合平均分

```
SELECT student, subject, AVG(score)::numeric(10,2)
FROM test_scores
WHERE student IN ('leo','regina')
GROUP BY ROLLUP (student,subject)
ORDER BY student NULLS LAST, subject NULLS LAST;

 student |  subject  | avg
---------+-----------+-------
 leo     | algebra   | 82.00
 leo     | calculus  | 65.50
 leo     | chemistry | 75.50
 leo     | physics   | 72.00
 leo     | NULL      | 73.75
 regina  | algebra   | 72.50
 regina  | calculus  | 64.50
 regina  | chemistry | 73.50
 regina  | economics | 90.00
 regina  | physics   | 84.00
 regina  | NULL      | 75.44
 NULL    | NULL      | 74.65
(12 rows)
```

如果把 ROLLUP 的入参字段顺序反过来，那么得到的是每个学生分科目的平均分、每个科目的平均分、全局综合平均分，如示例 7-46 所示。

示例 7-46　统计每个学生分科目的平均分、每个科目的平均分、全局综合平均分

```
SELECT student, subject, AVG(score)::numeric(10,2)
FROM test_scores
WHERE student IN ('leo','regina')
GROUP BY ROLLUP (subject,student)
ORDER BY student NULLS LAST, subject NULLS LAST;

 student |  subject  | avg
---------+-----------+-------
```

```
 leo     | algebra   | 82.00
 leo     | calculus  | 65.50
 leo     | chemistry | 75.50
 leo     | physics   | 72.00
 regina  | algebra   | 72.50
 regina  | calculus  | 64.50
 regina  | chemistry | 73.50
 regina  | economics | 90.00
 regina  | physics   | 84.00
 NULL    | algebra   | 77.25
 NULL    | calculus  | 65.00
 NULL    | chemistry | 74.50
 NULL    | economics | 90.00
 NULL    | physics   | 78.00
 NULL    | NULL      | 74.65
(15 rows)
```

如果希望把前两种统计的结果综合一下，即包含每个学生所有科目的综合平均分、每个科目所有学生的综合平均分、每个学生每个科目的平均分以及全局不分科目和学生的综合平均分，那么可以写成 GROUPING SETS((student), (student, subject), (subject), ())，或者可以使用 CUBE(student, subject) 语法，如示例 7-47 所示。

示例 7-47　统计每个学生所有科目的综合平均分、每个科目所有学生的综合平均分、每个学生每个科目的平均分以及全局不分科目和学生的综合平均分

```
SELECT student, subject, AVG(score)::numeric(10,2)
FROM test_scores
WHERE student IN ('leo','regina')
GROUP BY CUBE (student, subject)
ORDER BY student NULLS LAST, subject NULLS LAST;

student  |  subject  | avg
---------+-----------+-------
 leo     | algebra   | 82.00
 leo     | calculus  | 65.50
 leo     | chemistry | 75.50
 leo     | physics   | 72.00
 leo     | NULL      | 73.75
 regina  | algebra   | 72.50
 regina  | calculus  | 64.50
 regina  | chemistry | 73.50
 regina  | economics | 90.00
 regina  | physics   | 84.00
 regina  | NULL      | 75.44
 NULL    | algebra   | 77.25
 NULL    | calculus  | 65.00
 NULL    | chemistry | 74.50
 NULL    | economics | 90.00
 NULL    | physics   | 78.00
 NULL    | NULL      | 74.65
(17 rows)
```

第 8 章
函数编写

PostgreSQL 同大多数数据库一样，可以把若干 SQL 语句组合在一起，然后将其作为一个单元来处理，并且每次运行时可以输入不同的参数。这种机制在不同数据库中的名称不一样，有的叫存储过程，有的叫用户自定义函数，而 PostgreSQL 统一称之为**函数**。

函数不是仅仅将一堆 SQL 语句编排在一起即可，其中还需要使用过程式语言（procedural language，PL）来对 SQL 语句的执行过程进行控制。在 PostgreSQL 中，你可以选择使用不同的语言来编写函数，而且可选择的语言有很多，其中 SQL、C、PL/pgSQL、PL/Perl 以及 PL/Python 一般都会随 PostgreSQL 安装包附带。此外还支持使用 PL/V8 语言，通过它你可以使用 JavaScript 语言来编写函数。PL/V8 是 Web 开发人员的最爱，因为 JSON、JSONB 数据类型和 JavaScript 语言是绝配。关于 JSON 和 JSONB 数据类型的详情，请参考 5.6 节。

你还可以按需安装 PL/R、PL/Java、PL/sh、PL/TSQL 等语言扩展，此外还有一些用于高端数据处理以及人工智能处理的试验性语言，比如 PL/Scheme 和 PL/OpenCL 等。你可以在官方手册的“过程式语言”一节中查到 PostgreSQL 支持的完整语言列表。

8.1 PostgreSQL函数功能剖析

PostgreSQL 的函数可分为基本函数、聚合函数、窗口函数和触发器函数四大类。我们首先介绍关于函数的基础知识，然后再详细介绍前述每类函数的具体特性。

8.1.1 函数功能基础知识介绍

在 PostgreSQL 中，不管你选择使用何种编程语言，所编写出来的函数的结构都是类似的，如示例 8-1 所示。

示例 8-1 函数的基本结构

```
CREATE OR REPLACE FUNCTION func_name(arg1 arg1_datatype DEFAULT arg1_default)
RETURNS some type | set of some type | TABLE (..) AS
$$
BODY of function
$$
LANGUAGE language_of_function
```

参数可以有默认值。函数调用者可以忽略有默认值的参数，即不用为其输入值，直接采用默认值即可。在函数定义中，可选参数必须排列在必选参数的后面。

入参支持命名参数和匿名参数两种形式，前者必须为参数起个名字，而后者不需要。我们建议使用命名参数，因为这样可以在函数体内通过参数名引用它，非常方便和直观。假设我们定义了一个可以接受三个入参的函数（其中两个入参是可选的）如下：

```
big_elephant(ear_size numeric, skin_color text DEFAULT 'blue',
name text DEFAULT 'Dumbo')
```

你可以在函数体内部通过参数名引用这些入参（`ear_size`、`skin_color` 等）。如果参数是匿名的，那么只能通过序列号的方式来访问它：$1、$2 和 $3。

如果使用了命名参数，那么调用函数时还可以采用以下带入参名的调用方式，该方式的好处是参数输入顺序与定义时的顺序不必完全相同：

```
big_elephant(name => 'Wooly', ear_size => 1.2)
```

即使函数定义时使用了命名入参，也可以按照参数位置来输入而不必带上参数的名字，比如 `big_elephant(1.2, 'blue', 'Wooly')`。那么什么时候需要使用带入参名的调用方式呢？如果函数有多个入参，并且其中大部分都是可选的，那么此时适合使用带参数名的调用方式。该方式可以实现覆盖参数默认值，而且参数的输入顺序与定义时的顺序不必相同。以上面对 `big_elephant` 函数的调用为例，`skin_color` 参数未出现，因此会取默认值 `'blue'`，`name` 参数的值会覆盖默认值，而且 `name` 定义时在最后，调用时却放在了第一个。如果按参数位置的形式来调用，那么要想给 `name` 赋值以覆盖其默认值的话，`skin_color` 参数就必须要显式输入。



在 PostgreSQL 9.5 以及更高的版本中，带参数名的调用语法是类似 `name => 'Wooly'` 这样的，但是在 PostgreSQL 9.4 以及之前的版本中，语法是 `name := 'Wooly'`。为保证前向兼容，`arg1_name := arg1_value` 这种语法在 PostgreSQL 9.5 及之后的版本中仍是支持的，但不建议继续使用，因为将来会不再支持。

定义函数时可以添加一些标记符来优化执行效率或者提升安全性，支持的标记符如下。

`LANGUAGE`**（使用的编程语言）**

指明本函数使用的编程语言，当然该语言必须在当前函数所在的 database 中已安装。执行 `SELECT lanname FROM pg_language;` 即可查到已安装的语言列表。

`VOLATILITY`**（结果的稳定性）**

该标记符可以告诉查询规划器，当该函数执行完毕后，得到的结果是否可以缓存下来以供下次使用。它有以下几个可选值。

`IMMUTABLE`（**结果恒定不变**）

任何情况下，只要调用该函数时使用相同的输入，就总会得到相同的输出。也就是说，该函数的内部逻辑对外界完全无依赖。这类函数最典型的例子是数学计算函数。注意，定义函数索引时必须使用 `IMMUTABLE` 函数。

`STABLE`（**结果相对稳定**）

如果在同一个查询语句中多次调用该函数，则每次调用时只要使用相同的输入就总会得到相同的输出。也就是说，该函数的内部逻辑在当前 SQL 的上下文环境内是有恒定输出的。

`VOLATILE`（**结果不稳定**）

每次调用该函数得到的结果可能都不同，即便每次都使用相同的输入也是这样。那些更改数据的函数和那些依赖系统时间这类环境设置的函数就属于 `VOLATILE` 类型。该项也是默认值。

请注意，`VOLATILITY` 标记符仅仅是给规划器提供了一个提示信息，规划器并不一定会按照此设置来进行处理。如果函数被标记为 `VOLATILE`，那么规划器每次遇到此函数都会重新解析并重新执行一遍；如果被标记为别的类型，那么规划器也可能不会对其执行结果进行缓存，因为规划器可能认为重新计算一遍反而会更快。

`STRICT`（**严格模式**）

对于一个严格模式的函数来说，如果有任何输入为 `NULL`，则规划器根本不会执行这个函数，直接返回 `NULL`。如果未显式指定为 `STRICT` 模式，则函数默认都是非严格模式的。写函数时，务必慎用 `STRICT`，因为用了以后可能会导致规划器不使用索引。请参考我们的博文“STRICT on SQL Functions”以获取更多细节。

`COST`（**执行成本估计**）

这是标记函数中计算操作密集程度的一个相对度量值。如果使用的是 SQL 或 PL/pgSQL 语言，则该值为 `100`；如果使用 C 语言，则该值为 `1`。该值会影响到规划器执行 `WHERE` 子句中的函数时的优先级，也会影响到是否对此函数进行结果集缓存的可能性判定。此值越大，则规划器会认为执行该函数需要耗费的时间越多。

`ROWS`（**返回结果集的行数估计**）

仅当函数返回的是一个结果集时，此标记符才有用。该值是返回的结果集中记录数的一个估计值。规划器会利用此数值来为此函数分析得出最佳的执行策略。

`SECURITY DEFINER`（**安全控制符**）

如果设置了安全控制符，则会以创建此函数的用户的权限执行此函数；如果未设置，则会以调用此函数的用户的权限执行此函数。如果某用户对某张表没有操作权限而又需要操作该表，那么就可以让创建该表的用户提供一个带 `SECURITY DEFINER` 标识的函数来对此表进行操作。可以看出，当需要进行表的访问权控制时，这个安全控制符还是很有用的。

`PARALLEL`（**并行度**）

该标记符是 PostgreSQL 9.6 新引入的。该标记表示允许规划器以并行模式运行。默认情况下，函数会被设置为 `PARALLEL UNSAFE`，这意味着任何调用该函数的语句都不会被分

布到多个工作进程上去并发执行。详情请参考官方手册“并行安全性”相关内容。支持的选项如下。

SAFE

该选项表示允许该函数被并行执行。如果函数是 IMMUTABLE 类型的，或者函数不更新数据或者不会修改事务状态或其他变量值，那么将其设为 SAFE 一般是没问题的。

UNSAFE

如果函数会修改非临时数据、访问序列号生成器或者事务状态，那么都应被设置为 UNSAFE。UNSAFE 的函数如果以并行模式执行可能会导致表数据被破坏或者其他系统状态被破坏，因此不允许被并行执行。

RESTRICTED

对于使用临时表、预解析语句或者客户端连接状态的函数可以使用该选项。设置为 RESTRICTED 的语句不会被禁止并行执行，但是它只能运行在并行组中的领导组（lead）中，也就是说该函数本身不会被并行执行，但它不会阻止调用它的 SQL 语句被并行执行。

本章的很多例子中都带有 PARALLEL 标记符，如果你的试验环境是 PostgreSQL 9.6 之前的版本，请在执行例子时把 PARALLEL 标记去掉。

8.1.2 触发器和触发器函数

任何一个功能健全的数据库都支持触发器功能。借助触发器机制，可以实现自动捕捉数据变化事件并进行相应处理。PostgreSQL 既支持对表创建触发器，也支持对视图创建触发器。

可以指定触发器在语句级或者记录级被触发。对语句级触发器来说，每执行一条 SQL 语句只会被触发一次；对记录级触发器来说，SQL 语句执行过程中每修改一条记录就会被触发一次。例如，假设你对某表执行了一个 UPDATE 语句，更新了 1500 条记录。那么该表上的语句级触发器只会触发一次，而记录级触发器会触发 1500 次。

你还可以更加精细地设置触发器的触发时机，系统支持 BEFORE、AFTER 以及 INSTEAD OF 这三种时机。BEFORE 类的触发器会在语句执行之前或者记录行被修改之前触发，你可以借此时机来取消此次修改或者对要修改的数据进行预先备份。AFTER 类的触发器会在语句执行之后或者记录行被修改之后触发，你可以借此时机来获得修改后的新值，该类触发器一般用于记录修改日志或者进行数据复制。INSTEAD OF 类的触发器会将原语句的操作内容替换掉。BEFORE 和 AFTER 类的触发器只能用于表，而 INSTEAD OF 类的触发器只能用于视图。

注意，如果要在触发器函数中进行数据修改，那么该触发器的触发时机一定只能是 BEFORE 类的，因为在 AFTER 阶段所有针对新记录的修改操作都会被忽略。

你还可以在定义触发器时加上 WHEN 条件，以限定只有那些满足筛选条件的记录被修改时才激活该触发器；也可以通过加上“UPDATE OF+ 字段列表”子句来指定只有修改了特定的列时才激活该触发器。如果希望更加深入细致地了解触发器与主体语句之间的触发联动机制，请参考 PostgreSQL 官方手册中的“触发器行为概览”一节的内容。我们还在示例 7-5 中演示了一个视图触发器的用法。

PostgreSQL 提供了一种专门用于处理触发器逻辑的函数，称为**触发器函数**，其行为模式与其他函数类似，内部的代码结构也相同。触发器函数与普通函数的唯一区别在于输入形参和输出类型。触发器函数从不需要参数，因为可以在函数内部访问数据并对其进行修改。

触发器函数的返回值永远是 `trigger` 类型。PostgreSQL 的触发器函数与其他函数类似，因此一个触发器函数可以被多个触发器共用。几乎没有哪家数据库能支持该特性，因为一般的数据库中都是把触发器和触发器函数作为一个完整的对象绑定在一起的，这样的触发器处理逻辑无法被别的触发器重用。

在 PostgreSQL 中，每个触发器有且仅有一个配套的触发器函数。如果由于业务需要必须将逻辑分散到多个触发器函数中，那么就得创建多个触发器来调用它们，这些触发器的触发事件可以相同也可以不同。如果触发事件相同的话，那么系统会将触发器名称按字典顺序进行排序，然后逐个触发。后一个触发器可以看到前一个触发器的修改结果。每一个触发器并不是一个独立的事务，因此如果在某个触发器中执行了回滚操作，那么在此触发器之前执行过的触发器修改都会被回滚掉。

你可以使用 PostgreSQL 支持的任何一种编程语言来编写触发器函数，但注意不能使用 SQL，因为它不是过程式语言。SQL 对应的过程式语言是 PL/pgSQL，它也是目前为止在 PostgreSQL 环境中使用最广泛的语言。8.3.2 节会演示如何使用 PL/pgSQL 编写触发器函数。

8.1.3 聚合操作

大多数其他数据库仅允许使用 ANSI SQL 标准中定义的那些聚合函数，比如 `MIN`、`MAX`、`AVG`、`SUM` 和 `COUNT` 等。在 PostgreSQL 中则无此限制，你可以自行实现比以上函数功能更复杂的聚合函数。在 PostgreSQL 中，一个聚合函数同时也可以作为窗口函数（相关概念请参见 7.3 节）来使用，因此你可以实现事半功倍的效果。

你可以使用 PostgreSQL 所支持的几乎任何语言（包括 SQL 语言在内）来编写聚合函数。聚合函数一般是基于一个或者多个子函数实现的。首先至少得有一个状态转换函数，该函数会反复执行多次，以将输入的多行记录聚合为一个单独的结果。你还可以建立用于处理初始状态和终结状态的函数，不过这两个函数是可选的。前述这几类函数都是聚合函数的子函数，它们可以使用不同的编程语言来实现，我们在"PostgreSQL Aggregates"这篇博文中演示了基于 PL/pgSQL、PL/Python 和 SQL 等多种语言的子函数构造而成的聚合函数示例。

不管你使用何种编程语言来编写这些子函数，最终将它们整合为一个聚合函数的语法是一样的，如下所示。

```
CREATE AGGREGATE my_agg (input data type) (
SFUNC=state function name,
STYPE=state type,
FINALFUNC=final function name,
INITCOND=initial state value, SORTOP=sort_operator
);
```

`SFUNC` 状态切换函数（这个名称不够直观，此处所谓的"状态"是指在聚合运算过程中每处理完一条记录后得到的中间结果）是实现聚合运算的逻辑主体，它会将自身上一次被调

用后生成的计算结果作为本次计算的输入，同时输入的还有当前新一条的待处理记录，这样将所有记录一条条累积处理完毕后，就得到了基于整个目标记录集的“状态”，也就是最终的聚合结果。有的情况下，SFUNC 处理得到的结果就是聚合函数需要的最终结果，但另外一些情况下，SFUNC 处理完毕的结果还需要再进行最终加工才是我们想要的聚合结果，FINALFUNC 就是负责这个最终加工步骤的函数。FINALFUNC 是可选的，由于它的作用是对 SFUNC 函数的输出结果做最后加工，因此它的输入一定是 SFUNC 函数的输出。INITCOND 也是可选的，如果设定了该条目，那么其值会被作为 SFUNC 函数的“状态”的初始值。

最后的 SORTOP 也是可选的，其值是类似于 > 或 < 这样的运算符，它的作用是为类似 MAX、MIN 这样的排序操作指定排序运算符。指定了 SORTOP 运算符后，规划器会使用索引来进行 MAX、MIN 这样的聚合运算，由于索引是有序的，所以可以快速定位到索引的头部或者尾部以寻找 MAX、MIN 值，这样就不需要对所有记录逐条进行大小值判断，整体运算速度也得以极大提升。不过 SORTOP 运算符的使用有一个先决条件，那就是在聚合运算的目标表上，以下两条语句的执行结果必须完全相同。

```
SELECT agg(col) FROM sometable;
SELECT col FROM sometable ORDER BY col USING sortop LIMIT 1;
```



在 PostgreSQL 9.4 中，CREATE AGGREGATE 语法得到了强化，新增了对移动窗口聚合函数的支持，该特性对于窗口可移动的窗口函数是很有意义的。详情请参考 9.4 版官方手册中的“创建聚合函数”一节的内容。



在 PostgreSQL 9.6 中，聚合函数也开始支持并行执行。通过设置函数的 parallel 属性来指定某个函数是否启用并行，具体可以设为 safe、unsafe、restricted 这几个值。如果不设，默认是 unsafe。除了 parallel 属性，还增加了 combinefunc、serialfunc 和 deserialfunc 这几个并行聚合相关的属性。详情请参考官方手册中的“SQL 创建聚合函数”一节的内容。

一般来说，聚合函数只针对一个列进行聚合运算，比如 MAX、MIN、AVG 等，但事实上完全可以创建针对多个列进行聚合运算的聚合函数。如果你需要这样的多列聚合函数，请参考“How to Create Multi-Column Aggregates”这篇博文来了解具体的实现方法。

前面已经介绍过聚合函数是可以使用 SQL 来编写的。SQL 是一种极其易用的语言，你不需要关心那些各式各样的流程控制语句（因为 SQL 中不支持），而且你可能对 SQL 已经很熟悉，因此上手更加简单。当编写聚合函数时，仅仅使用 SQL 就可以实现很强大的功能。我们将在 8.2.2 节中介绍相关内容。

8.1.4　受信与非受信语言

PostgreSQL 支持的函数语言可按照信任级别分为两类：受信语言与非受信语言。很多语言（但并不是所有）同时提供了受信与非受信版本。这里所说的“受信”是指该语言不可能对数据库服务器的底层操作系统造成任何破坏，这一点是通过拒绝它执行操作系统的高风险操作来保证的。简要介绍如下。

受信语言

受信语言不具备直接访问数据库服务器底层文件系统的权限，因此在该类语言中不能直接执行操作系统级命令。任何权限级别的用户都可以使用受信语言创建函数。包括 SQL、PL/pgSQL、PL/Perl 和 PL/V8 在内的语言都是受信语言。

非受信语言

非受信语言可以直接与操作系统进行交互，通过该类语言可以直接调用操作系统提供的函数和 Web 服务接口。PostgreSQL 中只有超级用户才有权使用非受信语言编写函数，但超级用户有权将基于非受信语言的函数的执行权限授予普通用户。一般来说，非受信语言的命名会以 U 结尾，比如 PL/PerlU、PL/PythonU 等。这一点并不绝对，比如 PL/R 就是个例外。

8.2 使用SQL语言来编写函数

大多数情况下，我们仅仅使用 SQL 语言来编写单个语句，但事实上它也可以用于编写函数。在 PostgreSQL 中，将现有的一条 SQL 语句改造为函数是一件又快又简单的事情：只需在现成的 SQL 基础上加上函数头和函数尾就可以了。但编写简单同时也意味着功能有限。SQL 不是一种过程式语言，因此你无法用上条件分支判断、循环或者定义变量等过程式语言的特性。此外还有一个更严重的限制，那就是无法执行使用函数入参动态拼装的 SQL 语句。

当然，SQL 函数也有其优点。查询规划器可以深入到 SQL 函数内部，并对其中每一条 SQL 语句进行分析和优化，该过程被称为 inlining，即**内联处理**。对于别的语言编写的函数，规划器只能将其当成黑盒处理。只有 SQL 函数可以被内联处理，这使得 SQL 函数能够充分利用索引并减少重复计算。

8.2.1 编写基本的SQL函数

示例 8-2 演示了一个最基本的 SQL 函数，该函数向表中插入一条记录，并返回一个标量值。

示例 8-2 创建一个 SQL 函数，其返回值为新插入的记录的唯一 ID

```
CREATE OR REPLACE FUNCTION write_to_log(param_user_name varchar,
param_description text)
RETURNS integer AS
$$
INSERT INTO logs(user_name, description) VALUES($1, $2)
RETURNING log_id;
$$
LANGUAGE 'sql' VOLATILE;
```

函数的调用语法如下所示。

```
SELECT write_to_log('alex', 'Logged in at 11:59 AM.') As new_id;
```

类似地，也可以在 SQL 函数中更新数据并返回一个标量或者不返回，如示例 8-3 所示。

示例 8-3　创建一个进行更新操作的 SQL 函数

```
CREATE OR REPLACE FUNCTION
update_logs(log_id int, param_user_name varchar, param_description text)
RETURNS  void AS
$$
UPDATE logs SET user_name = $2, description = $3
 , log_ts = CURRENT_TIMESTAMP WHERE log_id = $1;
$$
LANGUAGE  'sql' VOLATILE;
```

通过以下语句来调用此函数。

```
SELECT update_logs(12, 'alex', 'Fell back asleep.');
```

基本上所有编程语言编写的函数都支持返回结果集，SQL 函数也不例外，它有三种返回结果集的方法：第一种是 ANSI SQL 标准中规定的 `RETURNS  TABLE` 语法，第二种是使用 `OUT` 形参，第三种是使用复合数据类型。其他数据库一般也都是基于 `RETURNS TABLE` 语法来实现结果集的返回。示例 8-4 演示了如何使用这三种方法来实现返回结果集。

示例 8-4　在函数中返回结果集

使用 `RETURNS TABLE` 语法的方式如下所示。

```
CREATE OR REPLACE FUNCTION select_logs_rt(param_user_name varchar)
RETURNS TABLE (log_id int, user_name varchar(50),
description text, log_ts timestamptz) AS
$$
SELECT log_id, user_name, description, log_ts FROM logs WHERE user_name = $1;
$$
LANGUAGE 'sql' STABLE PARALLEL SAFE;
```

使用 `OUT` 形参的方式如下所示。

```
CREATE OR REPLACE FUNCTION select_logs_out(param_user_name varchar, OUT log_id int
 , OUT user_name varchar, OUT description text, OUT log_ts timestamptz)
RETURNS SETOF record AS
$$
SELECT * FROM logs WHERE user_name = $1;
$$
LANGUAGE 'sql' STABLE PARALLEL SAFE;
```

使用复合数据类型的方式如下所示。

```
CREATE OR REPLACE FUNCTION select_logs_so(param_user_name varchar)
RETURNS SETOF logs AS
$$
SELECT * FROM logs WHERE user_name = $1;
$$
LANGUAGE 'sql' STABLE PARALLEL SAFE;
```

以上三种方式实现的函数的调用方法都是一样的。

```
SELECT * FROM select_logs_xxx('alex');
```

8.2.2　使用SQL语言编写聚合函数

是的！你没看错，PostgreSQL 支持用户在常见的 MIN、MAX、COUNT、AVG 等聚合函数之外自定义聚合函数。本节将演示如何使用 SQL 语言来创建一个用于计算几何平均值的聚合函数。**几何平均值**是指 *n* 个正数的连乘积的 *n* 次方根（(x1*x2*x3...xn)$^{(1/n)}$），它在金融、经济以及统计学领域有着广泛的应用。当样本数字的值域范围变化很大时，可以使用几何平均值来替代更常见的算术平均数。几何平均值可以使用更高效的公式来计算：EXP(SUM(LN(x))/n)，该公式使用了对数来将连续的乘法运算转换为连续的加法运算，因此计算机执行的效率更高。在下面的例子中，我们将使用该公式计算几何平均值。

为了构造几何平均值聚合函数，需要创建两个子函数：一个状态转换函数，用于把对数运算结果相加（参见示例 8-5）；一个最终处理函数，用于对对数之和进行取幂运算。此外还需要指定状态初始值为 0。

示例 8-5　创建几何平均值聚合函数的状态切换函数

```
CREATE OR REPLACE FUNCTION geom_mean_state(prev numeric[2], next numeric)
RETURNS numeric[2] AS
$$
SELECT
  CASE
    WHEN $2 IS NULL OR $2 = 0 THEN $1
    ELSE ARRAY[COALESCE($1[1],0) + ln($2), $1[2] + 1]
  END;
$$
LANGUAGE sql IMMUTABLE PARALLEL SAFE;
```

此处定义的状态切换函数有两个输入项：第一个是前次调用本状态切换函数计算后得到的结果，其类型为含两个元素的数字型数组；第二个是本轮计算要处理的样本值。如果第二个参数的值为 NULL 或者为 0，则本轮无须计算，直接返回参数 1 的值；否则将本次处理的样本数字的 ln 对数值累加到参数数组的第一个元素上，并对参数数组的第二个元素值加 1。这样最终得到结果就是含所有样本数字的 ln 对数值的总和以及总运算次数。

此外还需要一个如示例 8-6 所示的最终处理函数，该函数中需要将状态转换函数计算得到的两个值相除。

示例 8-6　创建几何平均值聚合函数的最终处理函数

```
CREATE OR REPLACE FUNCTION geom_mean_final(numeric[2])
RETURNS numeric AS
$$
SELECT CASE WHEN $1[2] > 0 THEN exp($1[1]/$1[2]) ELSE 0 END;
$$
LANGUAGE sql IMMUTABLE PARALLEL SAFE;
```

最后，需要将前面定义的这些子函数整合到一起，组成一个完整的聚合函数，语法如示例 8-7 所示。（请注意，本例中的聚合运算需要一个初始值 (0,0)，该初始值的类型与 SFUNC 的参数类型一定是一致的。）

示例 8-7　基于定义好的子函数来创建几何平均值聚合函数

```
CREATE AGGREGATE geom_mean(numeric) (
SFUNC=geom_mean_state,
STYPE=numeric[],
FINALFUNC=geom_mean_final,
PARALLEL = safe,
INITCOND='{0,0}'
);
```

接下来测试一下刚刚创建好的函数。在示例 8-8 中，我们计算出了马萨诸塞州各县的种族多样性排名，并列出种族多样性最好的 5 个县的数据。

示例 8-8　基于几何平均值来统计出种族多样性最好的 5 个县

```
SELECT left(tract_id,5) As county, geom_mean(val) As div_county
FROM census.vw_facts
WHERE category = 'Population' AND short_name != 'white_alone'
GROUP BY county
ORDER BY div_county DESC LIMIT 5;

county |     div_county
-------+---------------------
25025  | 85.1549046212833364
25013  | 79.5972921427888918
25017  | 74.7697097102419689
25021  | 73.8824162064128504
25027  | 73.5955049035237656
```

接下来我们步子迈大一点，直接将上面定义的聚合函数当作窗口函数来试一下，看效果如何，如示例 8-9 所示。

示例 8-9　列出 5 个种族多样性最好的人口普查区

```
WITH X AS (SELECT
  tract_id,
  left(tract_id,5) As county,
  geom_mean(val) OVER (PARTITION BY tract_id) As div_tract,
  ROW_NUMBER() OVER (PARTITION BY tract_id) As rn,
  geom_mean(val) OVER(PARTITION BY left(tract_id,5)) As div_county
FROM census.vw_facts WHERE category = 'Population' AND short_name != 'white_alone'
)
SELECT tract_id, county, div_tract, div_county
FROM X
WHERE rn = 1
ORDER BY div_tract DESC, div_county DESC LIMIT 5;

tract_id    | county |    div_tract         |    div_county
------------+--------+----------------------+------------------
25025160101 | 25025  | 302.6815688785928786 | 85.1549046212833364
25027731900 | 25027  | 265.6136902148147729 | 73.5955049035237656
25021416200 | 25021  | 261.9351057509603296 | 73.8824162064128504
25025130406 | 25025  | 260.3241378371627137 | 85.1549046212833364
25017342500 | 25017  | 257.4671462282508267 | 74.7697097102419689
```

8.3 使用PL/pgSQL语言编写函数

如果 SQL 语言已经不能满足你编写函数的需求，一般来说常见的解决方案是转为使用 PL/pgSQL。PL/pgSQL 优于 SQL 的地方在于，它支持通过 `DECLARE` 语法定义本地变量以及支持流程控制语法。

8.3.1 编写基础的PL/pgSQL函数

为了展示 PL/pgSQL 与 SQL 的语法区别，我们在示例 8-10 中用 PL/pgSQL 重写了示例 8-4 中的函数例子。

示例 8-10 使用 PL/pgSQL 编写返回值为表类型的函数

```
CREATE FUNCTION select_logs_rt(param_user_name varchar)
RETURNS TABLE (log_id int, user_name varchar(50),
description text, log_ts timestamptz) AS
$$
BEGIN
        RETURN QUERY
    SELECT log_id, user_name, description, log_ts FROM logs
     WHERE user_name = param_user_name;
END;
$$
LANGUAGE 'plpgsql' STABLE;
```

8.3.2 使用PL/pgSQL编写触发器函数

由于 PostgreSQL 不支持使用 SQL 编写触发器函数，因此 PL/pgSQL 就成了编写触发器函数的首选。本节将介绍如何使用 PL/pgSQL 编写基本的触发器函数。

总共需要两个步骤：第一步是写一个触发器函数，第二步是将此触发器函数显式附加到合适的触发器上。第二步将处理触发器的函数与触发器本身分离开，这是 PostgreSQL 的一个强大的功能。你可以将同一个触发器函数附加到多个触发器上，从而实现触发器函数逻辑的重用。该模式是 PostgreSQL 的独创功能，没有任何别的数据库支持该特性。由于触发器函数之间是完全独立的，你可以为每个触发器函数选择不同的编程语言，这些不同语言编写的触发器完全可以协同工作。PostgreSQL 支持通过一个触发事件（`INSERT`、`UPDATE`、`DELETE`）激活多个触发器，而且每个触发器可以基于不同的语言编写。例如，假设数据库中发生某一事件时你需要将其记录下来，另外还需要发邮件通知你。那么你可以使用 PL/PythonU 或者 PL/PerlU 语言编写一个具备发送邮件功能的触发器；同时可以使用 PL/pgSQL 语言编写一个记录日志的触发器。发生指定事件时，这两个触发器会同时被触发，执行各自的任务。

示例 8-11 中演示了如何创建一个基本的触发器函数及配套的触发器。

示例 8-11　通过触发器对新插入的记录或者修改的记录打时间戳

```
CREATE OR REPLACE FUNCTION trig_time_stamper() RETURNS trigger AS ❶
$$
BEGIN
    NEW.upd_ts := CURRENT_TIMESTAMP;
    RETURN NEW;
```

```
END;
$$
LANGUAGE plpgsql VOLATILE;

CREATE TRIGGER trig_1
BEFORE INSERT OR UPDATE OF session_state, session_id ❷
ON web_sessions
FOR EACH ROW EXECUTE PROCEDURE trig_time_stamper();
```

❶ 定义触发器函数。该函数适用于任何带有 `upd_ts` 字段的表。该函数会先将 `upd_ts` 字段的值更新为当前时间戳，然后再返回修改后的记录。

❷ “字段级触发”是 9.0 版开始支持的一个特性，通过该特性可以将触发器的触发时机精确到字段级别。在 9.0 版之前，只要发生了 `UPDATE` 或者 `INSERT` 动作，上面示例中的触发器都会被触发。因此，如果要实现字段级触发控制，就必须拿 `OLD.some_column` 和 `NEW.some_column` 进行对比，找到发生变化的字段，然后才能判定是否要进行“字段级触发”。（请注意：`INSTEAD OF` 触发器不支持该特性。）

8.4 使用PL/Python语言编写函数

Python 是一种非常灵活的语言，它支持非常丰富的功能扩展库。据我们所知，PostgreSQL 是唯一一种允许用户使用 Python 语言来编写函数的数据库。从 9.0 版开始，PostgreSQL 还同时支持了 Python 2 和 Python 3 两种语言。



你可以在同一个 database 中同时安装 PL/Python2U 和 PL/Python3U 这两个语言包，但在同一个用户会话上不能同时使用这两种语言。这就意味着你不能在同一个语句中同时调用分别由 PL/Python2U 和 PL/Python3U 编写的函数。你在系统中会见到一种叫作 PL/PythonU 的语言，它实际上是系统为了保持前向兼容而为 PL/Python2U 语言创建的一个别名。

在使用 PL/Python 语言之前，要先在服务器上搭建好 Python 运行环境。Windows 和 Mac 平台的 Python 安装包可以从 http://www.python.org/download/ 站点下载。Linux/Unix 平台的各种发行版上一般都已经附带了 Python 环境，因此无须额外安装。请参考 PostgreSQL 官方手册中对 PL/Python 的相关介绍来了解详情。搭建好 Python 运行环境之后，需要为 PostgreSQL 安装 Python 语言扩展包。

```
CREATE EXTENSION plpython2u;
CREATE EXTENSION plpython3u;
```

在 PostgreSQL 上安装 Python 语言扩展包之前，务必确保服务器操作系统上的 Python 运行环境已经正常，否则你可能会遇到各种奇奇怪怪的问题。

由于 PostgreSQL 的 PL/PythonU 语言扩展包是基于某个具体版本的 Python 语言包编译出来的，因此你需要保证服务器上的 Python 版本与 `plpythonu` 扩展包的版本是匹配的。例如，假设你的 `plpython2u` 扩展包是基于 Python 2.7 编译的，那么服务器上就需要安装好 Python 2.7 运行环境。

编写基本的Python函数

PostgreSQL 会自动在 PostgreSQL 数据类型与 Python 数据类型间进行双向转换。PL/Python 语言编写的函数支持返回数组和复合数据类型。你可以使用 PL/Python 来编写触发器函数和聚合函数。我们在 PostgresOnline 站点上提供了一系列介绍 PL/Python 的文章，其中有相关的语法示例。

Python 语言可以实现一些通过 PL/pgSQL 语言无法实现的功能。示例 8-12 中演示了如何使用 PL/Python 语言来编写一个文本搜索函数，该函数可以实现对 PostgreSQL 在线官方手册的内容进行检索。

示例 8-12　使用 PL/Python 语言编写的函数来搜索 PostgreSQL 官方手册的内容

```
CREATE OR REPLACE FUNCTION postgresql_help_search(param_search text)
RETURNS text AS
$$
import urllib, re ❶
response = urllib.urlopen(
    'http://www.postgresql.org/search/?u=%2Fdocs%2Fcurrent%2F&q=' + param_search
) ❷
raw_html = response.read() ❸
result =
        raw_html[raw_html.find("<!-- docbot goes here -->") :
        raw_html.find("<!-- pgContentWrap -->") - 1] ❹
result = re.sub('<[^<]+?>', '', result).strip() ❺
return result ❻
$$
LANGUAGE plpython2u SECURITY DEFINER STABLE;
```

❶ 导入接下来需要使用的功能库。

❷ 在连接搜索词之后执行搜索。

❸ 读取返回的搜索结果并将其保存到一个名为 `raw_html` 的变量中。

❹ 从 `raw_html` 中将 `<!-- docbot goes here -->` 和 `<!-- pgContentWrap -->` 之间包含的内容截取出来，并存放到一个名为 `result` 的新变量中。

❺ 将 `result` 开头和结尾的 HTML 标记和空格删除。

❻ 返回 `result` 变量的内容。

调用 Python 函数与调用别的语言编写的函数没什么两样。在示例 8-13 中，我们使用在示例 8-12 中创建的函数来搜索三个字符串。

示例 8-13　在查询语句中使用 Python 函数

```
SELECT search_term, left(postgresql_help_search(search_term),125) As result
FROM (VALUES ('regexp_match'),('pg_trgm'),('tsvector')) As x(search_term);
```

前面提到过，PL/Python 是一种非受信语言，而且没有相应的受信版本。这意味着只有超级用户才能使用 PL/Python 编写函数，并且使用该语言编写出来的函数可以直接操作文件系统。示例 8-14 就利用了 PL/Python 的这种能力来获得一个目录中的文件列表。请注意，从操作系统的角度来看，PL/Python 函数是以 PostgreSQL 安装时创建的 `postgres` 操作系统账户身份来执行的，因此你在执行该示例之前需要确保 `postgres` 账户对该示例中使用的目录拥有访问权限。

示例 8-14　列出一个目录中的所有文件

```
CREATE OR REPLACE FUNCTION list_incoming_files()
RETURNS SETOF text AS
$$
import os
return os.listdir('/incoming')
$$
LANGUAGE 'plpython2u' VOLATILE SECURITY DEFINER;
```

可以通过以下语句执行上面创建的函数。

```
SELECT filename
FROM list_incoming_files() As filename
WHERE filename ILIKE '%.csv'
```

8.5　使用PL/V8、PL/CoffeeScript以及PL/LiveScript语言来编写函数

PL/V8（又名 PL/JavaScript）是一种基于 Google V8 引擎的受信语言。通过它可以用 JavaScript 来编写函数，并使用 JSON 数据类型来与外界交互。PL/V8 并不是 PostgreSQL 的一个核心功能，因此在比较流行的 PostgreSQL 发行版中都不附带此语言包。你可以通过源码自行编译安装。我们已经为你提供了编译好的 Windows 平台安装包。你可以从 PostgresOnline 站点下载到 PostgreSQL 9.6 版本的 PL/V8 安装包（含 32 位和 64 位版本）。

在 PostgreSQL 中安装了 PL/V8 扩展包后，你会发现新增支持的语言不是一种，而是三种，不过它们都是 JavaScript 的相关语言。

PL/V8（plv8）

　这是最基本的 JavaScript 语言，也是下面两种语言的基础。

PL/CoffeeScript（plcoffee）

　CoffeeScript 是一门简洁的、构架于 JavaScript 之上的预处理器语言，可以静态编译成 JavaScript。其语法类似于 Python，也使用了缩进格式来表达代码段之间的隶属关系，从而省掉了烦人的大括号。

PL/LiveScript（plls）

　LiveScript 是 CoffeeScript 语言的一个分支，其语法与 CoffeeScript 类似，但拥有更多的语法特性。"CoffeeScript: 10 Reasons to Switch from CoffeeScript to LiveScript"这篇文章中认为 LiveScript 是 CoffeeScript 的理想替代品。相比 CoffeeScript，LiveScript 拥有更多类似于 Python、F# 和 Haskell 的特性。如果你正在寻找一门比 PL/Python 占用内存空间更小的受信语言，那么应该试试 LiveScript。

PL/CoffeeScript 和 PL/LiveScript 语言都是基于相同版本的 PL/V8 库编译的，因此二者的功能本质上与 PL/V8 完全一致。事实上，如果这两种语言你试用之后觉得不合适，那么也可以很轻易地切换回 PL/V8。这三种语言都是受信语言，这意味着它们无法访问底层文件系统，但没有超级用户权限的用户可以用它们来实现函数。

示例 8-15 中是创建这三个语言包的命令。如果你需要在某个 database 中使用这些语言，那么就必须在其中执行一遍这些安装命令。这三种语言可以单独按需安装。

示例 8-15　PL/V8 系列语言包的安装

```
CREATE EXTENSION plv8;
CREATE EXTENSION plcoffee;
CREATE EXTENSION plls;
```

与 PL/pgSQL 相比，上述的 PL/V8 系列语言能够提供很多关键的功能特性，这使得它们有着独特的存在价值。这些特性中的一部分只有 PL/R 这种高端过程式语言才支持。

- 与 SQL 和 PL/pgSQL 相比，数学运算速度更快。
- 创建窗口函数的能力。SQL、PL/pgSQL、PL/Python 均不支持该能力（但 PL/R 和 C 语言是支持的）。
- 创建触发器函数和聚合函数的能力。
- 支持语句预解析、子事务、内嵌函数、类以及 try-catch 异常处理机制。
- 使用 `eval` 函数动态执行 JavaScript 代码的能力。
- 支持 JSON 数据类型，能对 JSON 对象进行循环筛选处理。
- 在 `DO` 命令的匿名代码块中访问函数的能力。
- 兼容 Node.js。PL/V8 和 Node.js 均使用了谷歌 V8 引擎，因此很多适用于 Node.js 的库可以不经修改就直接应用于 PL/V8。这给 Node.js 开发人员以及其他使用 JavaScript 进行网络应用开发的人们带来了很多便利。PostgreSQL 中有一个名为 `plv8x` 的扩展包，它使得 Node.js 的库在 PL/V8 中重用起来更加容易。

PostgresOnline 博客站点上提供了一些介绍 PL/V8 用法的例子。在有的例子里，我们从网上找来了大块的 JavaScript 代码并修改移植为 PL/V8 语言，详情可参考“Using PLV8 to Build JSON Selectors”这篇博文。前述 PL/V8 系列语言可完美地辅助 Web 应用开发，因为大量的客户端 JavaScript 代码都是可以直接拿来重用的。不过，对于 PostgreSQL 来说，这几种语言更为重要的意义在于它们都是全功能的强大语言，可用于处理数值计算、数据修改以及很多其他任务。

8.5.1　编写基本的函数

PL/V8 语言的主要优点之一就是可以在 PL/V8 函数中直接调用任何 JavaScript 函数，而且几乎不需要做修改。例如，网上有很多对电子邮件地址进行合法性验证的 JavaScript 函数，我们随便找了一个并将其封装为 PL/V8 函数，如示例 8-16 所示。

示例 8-16　使用 PL/V8 函数来验证电子邮件地址的合法性

```
CREATE OR REPLACE FUNCTION
validate_email(email text) returns boolean as
$$
 var re = /\S+@\S+\.\S+/;
 return re.test(email);
$$ LANGUAGE plv8 IMMUTABLE STRICT PARALLEL SAFE;
```

上面的例子中使用了一个 JavaScript 正则表达式对象来检查电子邮件地址的合法性。示例 8-17 中演示了如何使用此函数。

示例 8-17　调用 PL/V8 语言编写的电子邮件地址合法性校验函数

```
SELECT email, validate_email(email) AS is_valid
 FROM (VALUES ('alexgomezq@gmail.com')
 ,('alexgomezqgmail.com'),('alexgomezq@gmailcom')) AS x (email);
```

输出结果如下：

```
       email          | is_valid
----------------------+----------
alexgomezq@gmail.com  | t
alexgomezqgmail.com   | f
alexgomezq@gmailcom   | f
```

虽然可以使用 PL/pgSQL 语言以及 PostgreSQL 自己的正则表达式功能来实现与上例中完全相同的验证功能，但我们刚刚还是光明正大地拿来了一段别人的成熟代码，然后没花任何时间就实现了预期的功能。对开发人员来说，这难道不是最理想的人生体验吗？如果你是一名 Web 开发人员，而且需要在客户端和数据库服务器端同时对某份数据进行逻辑完全相同的合法性验证，那么使用 PL/V8 可以使你的工作事半功倍，只要在客户端写好逻辑，然后复制粘贴到数据库端就可以了。

你可以创建一张带一个 `text` 字段的表（该 `text` 字段用于存储 JavaScript 函数），然后将这些校验函数都存入该表。然后通过一些设置，可以实现 PostgreSQL 启动时自动加载表中存储的这些函数，此后在数据库的任何 PL/V8 函数中都可以随便调用这些函数了。具体的操作步骤请参考 Andrew Dunstan 的博文“Loading Useful Modules in PLV8”。能够实现 JavaScript 函数自动加载的关键在于，PL/V8 原生支持了 `eval` 函数，通过该函数可以动态执行任何 JavaScript 命令，因此才得以在启动阶段就对函数进行预加载。

我们通过一个在线语法转换器（js2coffee.org）将示例 8-17 中的 JavaScript 函数转换成了基于 CoffeeScript 语法的函数，如示例 8-18 所示。

示例 8-18　使用 PL/Coffee 语言编写的电子邮件地址校验函数

```
CREATE OR REPLACE FUNCTION
validate_email(email text) returns boolean as
$$
    re = /\S+@\S+\.\S+/
    return re.test email
$$
LANGUAGE plcoffee IMMUTABLE STRICT PARALLEL SAFE;
```

CoffeeScript 与 JavaScript 之间的语法差别并不大，主要的变化是去掉了小括号、大括号和分号。LiveScript 的语法和 CoffeeScript 的语法几乎完全一样，唯一的差别就是语言声明要改为 `LANGUAGE plls`。

8.5.2　使用PL/V8来编写聚合函数

在示例 8-19 和示例 8-20 中，我们使用 PL/V8 语言重写了计算几何平均值的聚合函数所使用的状态切换函数和最终处理函数（原例子请参考 8.2.2 节）。

示例 8-19　PL/V8 版的几何平均值聚合函数的状态切换函数

```
CREATE OR REPLACE FUNCTION geom_mean_state(prev numeric[2], next numeric)
RETURNS numeric[2] AS
$$
    return (next == null || next== 0) ? prev :
    [(prev[0] == null)? 0: prev[0] + Math.log(next), prev[1] + 1];
$$
LANGUAGE plv8 IMMUTABLE PARALLEL SAFE;
```

示例 8-20　PL/V8 版的几何平均值聚合函数的最终处理函数

```
CREATE OR REPLACE FUNCTION geom_mean_final(in_num numeric[2])
RETURNS numeric AS
$$
  return in_num[1] > 0 ? Math.exp(in_num[0]/in_num[1]) : 0;
$$
LANGUAGE plv8 IMMUTABLE PARALLEL SAFE;
```

最后，`CREATE AGGREGATE` 命令将各子函数整合成我们想要的聚合函数。不管子函数采用什么语言编写，此处的语法都是一样的，具体如示例 8-21 所示。

示例 8-21　PL/V8 版的几何平均值聚合函数的最终定义

```
CREATE AGGREGATE geom_mean(numeric) (
  SFUNC=geom_mean_state,
  STYPE=numeric[],
  FINALFUNC=geom_mean_final,
  PARALLEL = safe,
  INITCOND='{0,0}'
);
```

你可以再运行一遍示例 8-9，但把其中的 `geom_mean` 函数换为此处的 PL/V8 版本。得到的结果与当初使用 SQL 版本 `geom_mean` 计算得到的结果肯定是一样的，但 PL/V8 版本的运算速度比 SQL 版本要快两到三倍。对于数学运算来说，你会发现在很多情况下，用 PL/V8 语言编写的函数要比用 SQL 编写的功能相同的函数快 10 到 20 倍。

8.5.3　使用PL/V8编写窗口函数

PostgreSQL 有很多内置的窗口函数，7.3 节中已讨论过。任何一个聚合函数，包括用户自定义的聚合函数，都可以用作窗口聚合函数。仅这两个功能点已足以使 PostgreSQL 在所有的关系型数据库中脱颖而出。更加令人印象深刻的是，PostgreSQL 还支持用户自定义窗口函数。

不过请注意，PostgreSQL 所支持的大多数过程式语言都不能用于创建窗口函数。不管是内置的 PL/pgSQL 还是 SQL，抑或是 PL/Python 和 PL/Perl 都不支持该特性。C 语言可以，但需要编译，这很麻烦。PL/R 部分支持该特性，但 PL/V8 不但完全支持创建自定义窗口函数，而且速度还很快（在很多情况下速度与用 C 编写是一样的），另外它还不需要像 C 一样对函数代码编译后才能用。

PL/V8 语言支持一个名为 `plv8.window_object` 的函数，该函数可以返回当前窗口对象的一个句柄，通过该句柄可以对窗口内的元素进行检视和访问。正是由于该特性的存在，PL/V8 才可以支持创建自定义窗口函数。

在示例 8-22 中，我们创建了一个窗口函数，其功能是对于表中每一行，判定它是否是一批“连续值记录”中的首记录，如果是就返回 true，否则返回 false。此处，“连续值记录”的判定标准是从该记录开始连续 N 条记录的特定字段值都相同。该函数可以让用户自行设定连续多少条记录值相同就算是“连续值记录”，ofs 参数用于设定该值。

示例 8-22　用 PL/V8 创建的窗口函数来标记重复记录值

```
CREATE FUNCTION run_begin(arg anyelement, ofs int) RETURNS boolean AS $$
  var winobj = plv8.get_window_object();
  var result = true;
  /** 获取当前值 **/
  var cval = winobj.get_func_arg_in_partition(0,
                                              0,
                                              winobj.SEEK_CURRENT,
                                              false);
  for (i = 1; i < ofs; i++){
       /** 获取下一个值 **/
       nval =  winobj.get_func_arg_in_partition(0,
                                                i,
                                                winobj.SEEK_CURRENT,
                                                false);
       result = (cval == nval) ? true : false;
       if (!result){
             break;
       }
       /** 下一个当前值是最后一个值 **/
       cval = nval;

  }
 return result;
$$ LANGUAGE plv8 WINDOW;
```

要将一个函数定义为窗口函数，必须在其末尾行加上 WINDOWS 标记符，如示例 8-22 所示。

在函数的主体逻辑中，必须要遍历访问窗口中的每条记录并对其进行计算和判定。如前所述，PL/V8 支持通过窗口句柄来实现这一功能，该句柄支持的所有 API 可在 PL/V8 官方手册的“PL/V8 窗口函数 API”一节中查到。我们在上面所编写的函数中会透过窗口查看当前记录的后 ofs 条记录的值。如果后 ofs 条记录的值与当前记录的值都一样，则说明当前记录是一批“连续值记录”的首记录，窗口函数返回 true，否则返回 false。PL/V8 提供了 get_func_arg_in_partiton 函数以扫描窗口中的记录。我们通过该函数查看该记录的后续记录，当判定到后续记录的值与当前记录值不同或者已到达最后一条记录时，则返回 false 并退出。

我们使用该函数来判定掷硬币游戏的赢家。每个玩家投四次硬币，如果连续三次都掷出头像面则算赢，如示例 8-23 所示。

示例 8-23　PL/V8 窗口函数的使用

```
SELECT id, player, toss,
 run_begin(toss,3) OVER (PARTITION BY player ORDER BY id) AS rb
    FROM coin_tosses
ORDER BY player, id;
```

```
id  | player | toss | rb
----+--------+------+----
 4  | alex   | H    | t
  8 | alex   | H    | t
 12 | alex   | H    | f
 16 | alex   | H    | f
  2 | leo    | T    | f
  6 | leo    | H    | f
 10 | leo    | H    | f
 14 | leo    | T    | f
  1 | regina | H    | f
  5 | regina | H    | f
  9 | regina | T    | f
 13 | regina | T    | f
  3 | sonia  | T    | t
  7 | sonia  | T    | t
 11 | sonia  | T    | f
 15 | sonia  | T    | f
(16 rows)
```

如需更多使用 PL/V8 编写的函数示例，请从 GitHub 上的“window regression script”项目下载。其中演示了如何使用 PL/V8 创建 PostgreSQL 内置的窗口函数（包括 `lead`、`lag`、`row_number`、`cume_dist`、`first_value`、`last_value` 等）。

第 9 章

查询性能调优

我们在使用数据库的过程中迟早会遇到语句性能问题，常见的解决方案是优化 SQL 语句本身的写法，辅以建立合适的索引以及更新规划器分析过程中所需的统计信息。为了帮助用户实现这些优化动作，PostgreSQL 提供了内置的执行计划解释器，通过它可以展示出一个 SQL 语句的执行计划。如果你了解如何编写正确的 SQL 语句、如何建立合适的索引，以及执行计划解释器的帮助，那么写出优秀的 SQL 语句并充分利用硬件的最大计算能力应该不是难事。

9.1 通过EXPLAIN命令查看语句执行计划

要定位语句的性能问题，最简单直接的方法就是使用 `EXPLAIN` 和 `EXPLAIN (ANALYZE)` 命令来分析其执行计划。PostgreSQL 从很早的版本开始就已经支持该命令，并且历年来其功能一直在不断地演进，目前已经非常成熟，可以展示出一个语句的执行计划方方面面的细节。在演进过程中，该命令支持的输出格式也越来越丰富。`EXPLAIN` 命令甚至支持将输出转储为 XML、JSON 或者 YAML 格式。

对于普通用户来说，该功能最令人激动的一次强化是几年前 pgAdmin 引入的图形化展示执行计划的能力。借助这种能力，你只需仔细观察执行计划图，即可了解语句的瓶颈点在哪里，哪些表应该建索引，以及实际的执行路径与预期的执行路径是否一致。

9.1.1 EXPLAIN选项

要执行非图形化的 `EXPLAIN` 分析，只需在 SQL 语句前加上 `EXPLAIN` 以及一些可选参数，然后再执行即可。

- `EXPLAIN` 本身的执行效果是输出执行计划而并不执行 SQL 语句本身。

- 加上 ANALYZE 参数之后（就像 EXPLAIN (ANALYZE)）的执行效果是执行该 SQL 语句本身，而且会将实际执行情况与执行计划进行对比分析，这可以用来评估执行计划的准确性。
- 在 EXPLAIN 后增加 VERBOSE 参数（语法是 EXPLAIN (VERBOSE)）将使得输出的执行计划步骤精确到列级别。
- 还有一个必须与 ANALYZE 参数联用的 BUFFERS 参数，其语法为 EXPLAIN (ANALYZE, BUFFERS)，通过它可以显示出执行计划过程中重用缓存数据时的命中次数，这个数字越大就表示本次查询过程中从内存缓存中获取的记录数越多，这些数据是之前的查询执行过程中缓存下来的，缓存中已有的数据块就不需要再从磁盘读取了。

完整的 SQL 语句执行计划解释语法是这样的：EXPLAIN (ANALYZE, VERBOSE, BUFFERS)+ 查询语句，执行后输出的结果包括执行时间、列的输出以及缓存命中次数等。

对于 UPDATE 或者 INSERT 这种 DML 语句来说，如果仅希望查看其通过 EXPLAIN (ANALYZE) 得到的执行计划而并不希望真正执行数据修改，可以把这个语句包装成一个事务块，即语句之前加 BEGIN，之后加 ROLLBACK。

可以通过 pgAdmin 这种图形化工具来实现图形化的 EXPLAIN。启动 pgAdmin 之后，请像平常一样编写好查询语句，但不要执行它，然后从下拉菜单中选择 EXPLAIN 或者 EXPLAIN (ANALYZE)。

9.1.2　运行示例以及输出内容解释

我们找个例子来试验一下。首先使用 EXPLAIN (ANALYZE) 命令，SQL 命令中使用的是示例 4-1 和示例 4-2 中创建的表。

我们想先测试语句不使用索引的情况，因此先将表上的主键删掉。

```
ALTER TABLE census.hisp_pop DROP CONSTRAINT IF EXISTS  hisp_pop_pkey;
```

删除表上所有的索引后，我们可以看到此时使用了最基础的执行计划，即全表扫描策略。如示例 9-1 所示。

示例 9-1　使用 EXPLAIN (ANALYZE) 查看全表扫描的执行计划

```
EXPLAIN (ANALYZE)
SELECT tract_id, hispanic_or_latino
FROM census.hisp_pop
WHERE tract_id = '25025010103';
```

如果仅使用 EXPLAIN 命令，则输出的结果是估算的执行计划成本。如果加上了 ANALYZE 参数，即使用 EXPLAIN (ANALYZE) 命令，则输出的分析结果中还会包含实际执行后得到的真实成本。

示例 9-2 是示例 9-1 的执行输出结果。

示例 9-2　EXPLAIN (ANALYZE) 的执行结果

```
Seq Scan on hisp_pop
    (cost=0.00..33.48 rows=1 width=16)
    (actual time=0.213..0.346 rows=1 loops=1)
    Filter: ((tract_id)::text = '25025010103'::text)
```

```
        Rows Removed by Filter: 1477
    Planning time: 0.095 ms
    Execution time: 0.381 ms
```

EXPLAIN 输出的执行计划会包含多个执行步骤。每一步都会有一个估算的执行成本范围，看起来像这样：cost=0.00..33.48，如示例 9-2 所示。本例中，第一个数字 0.00 是估算的该步骤的起始执行成本，第二个数字 33.84 是估算的该步骤的总执行成本。起始执行时间点之前会执行一些后续计算的准备动作，而读取数据、索引扫描、多表数据关联整合等动作都是在起始执行时间点之后发生的。如果执行方式为全表扫描，那么其起始执行成本为 0，因为这种场景下规划器只是简单地立即开始扫描全表数据，没有什么预备动作。

请注意，估算的执行成本值的单位并不是真实的时间单位，其单位取决于硬件环境以及执行成本相关的参数配置。因此，执行成本值仅具有相对意义，可用于比较同一台物理服务器上多个执行计划的效率。规划器的任务就是选出总体成本值最低的一个执行计划。

因为我们在示例 9-1 中加入了 ANALYZE 参数，规划器将真实地执行查询语句，所以执行时间中还会包含真正的执行时间统计。

通过示例 9-2 中的执行计划可以看到规划器选择了全表扫描策略，因为没有任何索引可用。下面输出的 Rows Removed by Filter:1477 是扫描过程中排除掉的不符合条件的记录数。

如果你使用的是 PostgreSQL 9.4 或者更高版本，EXPLAIN 输出的执行计划中还提供了执行计划分析时间和真正的执行时间。执行计划分析时间就是规划器分析出最终执行计划所消耗的时间；执行时间是按照执行计划执行并得到最终结果所用的时间。

我们把主键重新建起来：

```
ALTER TABLE census.hisp_pop ADD CONSTRAINT hisp_pop_pkey PRIMARY KEY(tract_id);
```

现在再次执行示例 9-1 的语句，得到的输出如示例 9-3 所示。

示例 9-3　使用了索引的 EXPLAIN (ANALYZE) 执行计划

```
Index Scan using idx_hisp_pop_tract_id_pat on hisp_pop
    (cost=0.28..8.29 rows=1 width=16)
    (actual time=0.018..0.019 rows=1 loops=1)
    Index Cond: ((tract_id)::text = '25025010103'::text)
Planning time: 0.110 ms
Execution time: 0.046 ms
```

此场景下规划器判定使用索引会比全表扫描效率更高，因此在执行计划中使用了索引扫描策略。估算的执行时间从 33.48 降为 8.29。起始执行成本也不再是 0，因为规划器需要先扫描索引，然后才能把命中的记录从磁盘取出来（如果所需数据已经存在于内存缓存中，也有可能是直接从内存取）。你也可以看到规划器不再需要扫描 1477 条记录，这极大地降低了执行成本。

对于如示例 9-4 所示的较复杂的查询，其执行计划中会包含更多的步骤，这些步骤也称为子执行计划。每一个子执行计划都有它自己的成本估算值，这些值会被累加到总执行计划的成本估算值中。父执行计划显示时总是排在最前面，其中记录的成本估算值和真实时间值就是其所有子计划相应项目值之和。子计划在显示时是按照其层级向右逐级缩进的。

示例 9-4　带 GROUP BY 和 SUM 的语句的 EXPLAIN (ANALYZE) 分析

```
EXPLAIN (ANALYZE)
SELECT left(tract_id,5) AS county_code, SUM(white_alone) As w
FROM census.hisp_pop
WHERE tract_id BETWEEN '25025000000' AND '25025999999'
GROUP BY county_code;
```

示例 9-5 中记录的是示例 9-4 中语句的执行计划，其中包含分组和求和操作。

示例 9-5　包含散列聚合策略的 EXPLAIN (ANALYZE) 分析结果

```
HashAggregate
    (cost=29.57..32.45 rows=192 width=16)
    (actual time=0.664..0.664 rows=1 loops=1)
    Group Key: "left"((tract_id)::text, 5)
    -> Bitmap Heap Scan on hisp_pop
        (cost=10.25..28.61 rows=192 width=16)
        (actual time=0.441..0.550 rows=204 loops=1)
        Recheck Cond:
            (((tract_id)::text >= '25025000000'::text) AND
            ((tract_id)::text <= '25025999999'::text))
        Heap Blocks: exact=15
        -> Bitmap Index Scan on hisp_pop_pkey
            (cost=0.00..10.20 rows=192 width=0)
            (actual time=0.421..0.421 rows=204 loops=1)
            Index Cond:
                (((tract_id)::text >= '25025000000'::text) AND
                ((tract_id)::text <= '25025999999'::text))
Planning time: 4.835 ms
Execution time: 0.732 ms
```

示例 9-5 中所示执行计划的顶层步骤是一个散列聚合操作。该操作包含一个位图表扫描子计划，该位图表扫描子计划又包含了一个位图索引扫描子计划。在本例中，因为我们是第一次执行此语句，所以执行计划分析时间远远超过了真正的执行时间。但 PostgreSQL 有执行计划以及数据缓存功能，所以如果我们再次执行该语句，或者执行一个可以共享缓存下来的执行计划的类似语句，那么执行计划的分析时间就会大大减少，而且真正的执行时间也可能会减少，因为该语句执行期间所需的很多数据可能已经缓存在内存中了。由于刚刚提到的这些缓存功能，我们第二次的运行时间统计数据如下：

```
Planning time: 0.200 ms
Execution time: 0.635 ms
```

9.1.3　图形化展示执行计划

如果你觉得阅读纯文字形式的执行计划是一件痛苦的事，那么图 9-1 中演示的图形化执行计划（对应 EXPLAIN (ANALYZE) 命令行）会解除你的烦恼。

你只需将鼠标放到图标上就能看到每个步骤的详细信息。

在结束本节之前，我们要向你介绍一个表格形式的执行计划展示工具（http://explain.depesz.com/），该工具由 Hubert Lubaczewski 创建，在此我们向他表示感谢。打开工具地址，然后将文本格式的执行计划复制过去，就可以得到一个格式化得非常漂亮的表格，如图 9-2 所示。

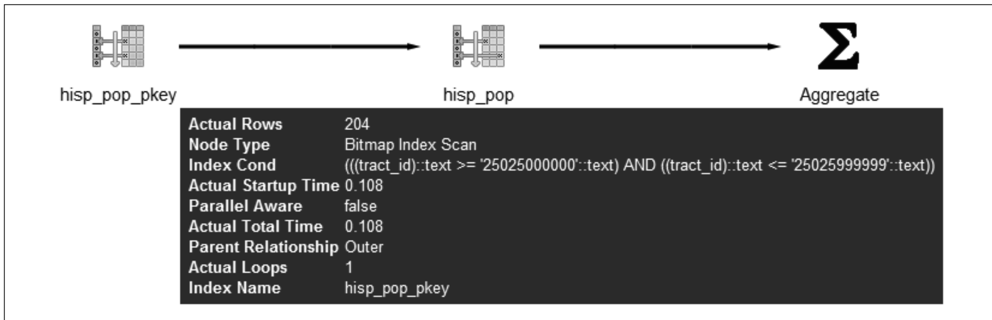


图 9-1：图形化展示执行计划

HTML TEXT STATS Did it help? Consider supporting us

Per node type stats

node type	count	sum of times	% of query
Bitmap Heap Scan	1	0.129 ms	19.4 %
Bitmap Index Scan	1	0.421 ms	63.4 %
HashAggregate	1	0.114 ms	17.2 %

Per table stats

Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
hisp_pop	1	0.129 ms	19.4 %
Bitmap Heap Scan	1	0.129 ms	100.0 %

图 9-2：在线执行计划分析工具

在输出的 HTML 表格中，你可以看到经过格式重排的带颜色分区的执行计划表，其中会以显眼的颜色高亮显示有问题的部分，如图 9-3 所示。表格中的 exclusive 列表示当前步骤的操作所耗时间，inclusive 列表示当前步骤及其所有子步骤的操作所耗时间。

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.114	0.664	↑ 192.0	1	1	→ HashAggregate (cost=29.57..32.45 rows=192 width=16) (actual time=0.664..0.664 rows=1 loops=1) Group Key: "left"((tract_id)::text, 5)
2.	0.129	0.550	↓ 1.1	204	1	→ Bitmap Heap Scan on hisp_pop (cost=10.25..28.61 rows=192 width=16) (actual time=0.441..0.550 rows=204 loops=1) Recheck Cond: (((tract_id)::text >= '25025000000'::text) AND ((tract_id)::text <= '25025999999'::text)) Heap Blocks: exact=15
3.	0.421	0.421	↓ 1.1	204	1	→ Bitmap Index Scan on hisp_pop_pkey (cost=0.00..10.20 rows=192 width=0) (actual time=0.421..0.421 rows=204 loops=1) Index Cond: (((tract_id)::text >= '25025000000'::text) AND ((tract_id)::text <= '25025999999'::text))

图 9-3：表格式的执行计划分析结果

尽管图 9-3 中 HTML 表格形式的执行计划提供的信息与纯文本形式的执行计划其实是一模一样的，但使用“彩色编码”和“分步骤操作时间统计”这两个功能，用户可以更轻松地对分析结果进行深入分析和挖掘。黄色、褐色以及红色的格子是潜在的性能瓶颈。

`rows x` 这一栏表示预估查询出的记录数，`rows` 栏显示的是执行完毕后实际查出的记录数。上表中显示的就是预估能返回 192 行记录，但实际仅返回 1 条。估算的记录数不准，一般是因为表的统计信息未及时更新所导致。所以经常性地对表进行分析操作是个好习惯，表的分析操作可以更新表上的统计信息，特别是刚刚对表进行过大规模的更新或者插入操作后，表分析操作很有必要。

9.2 搜集语句的执行统计信息

性能调优的第一步就是要确定哪些语句是性能瓶颈。PostgreSQL 提供了一个名为 `pg_stat_statements` 的性能监控扩展包以帮助用户找出耗时最长的语句。该扩展包能提供所有执行过的 SQL 语句的统计度量信息，包括哪些语句执行得最频繁以及每个语句的执行总耗时等。基于这些信息我们可以知道应将优化的重点放在哪里。

大多数 PostgreSQL 版本都自带 `pg_stat_statements` 扩展包，但启动时必须明确指定预加载其动态库，这样系统才会启动用于搜集统计数据的后台进程。设置方法如下所示。

(1) 在 postgresql.conf 配置文件中，将 `shared_preload_libraries = ''` 更改为 `shared_preload_libraries = 'pg_stat_statements'`。

(2) 在 postgresql.conf 文件的自定义选项部分，添加以下几行。

```
pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

(3) 重启 `postgresql` 服务。

(4) 登录到每一个希望进行 SQL 语句性能统计的 database 中并执行语句：`CREATE EXTENSION pg_stat_statements;`。

该扩展包提供了以下两个关键功能。

- 一个名为 `pg_stat_statements` 的视图，其中可以查询到当前登录用户在各 database 中执行过的所有 SQL 语句的统计信息。
- 一个名为 `pg_stat_statements_reset` 的函数，该函数可以将到目前为止的语句执行统计信息全部清空，不过只有超级用户才有权限执行此函数。

示例 9-6 中的语句可以查出 `postgresql_book` 这个 database 中最耗时的 5 个 SQL 语句。

示例 9-6　找出特定 database 中最耗时的语句

```
SELECT
    query, calls, total_time, rows,
    100.0*shared_blks_hit/NULLIF(shared_blks_hit+shared_blks_read,0) AS hit_percent
FROM pg_stat_statements As s INNER JOIN pg_database As d On d.oid = s.dbid
WHERE d.datname = 'postgresql_book'
ORDER BY total_time DESC LIMIT 5;
```

9.3　编写更好的SQL语句

最好也最简单的性能调优方法就是学会编写优秀的 SQL 语句。我们在大多数客户的项目中见过的 SQL 语句都写得不够好，它们未能发挥出 PostgreSQL 的真正威力。

写出的 SQL 语句很糟糕一般有两个主要原因。第一个原因是很多人会盲目地复用以前的 SQL 编写经验。例如，有人曾经写过一个使用了左连接的 SQL 语句并且执行效果还不错，那么他此后不管实际情况是什么样都一直使用左连接语法，但实际上，在有更多表参与关联运算的情况下，最好是使用内连接。与其他很多编程语言不一样，SQL 语言的编写经验不能盲目地复用。

第二个原因是人们对于最新的 SQL 语法特性一般无法及时跟进并学习了解。不要对 PostgreSQL 新版本中引入的那些新语法不以为意，它们可以简化开发，也可以让你少费脑筋，因此要积极地学习并利用起来。

要想编写出高效的 SQL 是需要很多练习的。只要你编写的 SQL 语句能得到正确结果，那么这个语句就不能算错，但其性能可能很差。本节中我们将指出人们常犯的一些错误。尽管本书是关于 PostgreSQL 的，但我们给出的这些建议其实也适用于其他关系型数据库。

9.3.1　在SELECT语句中滥用子查询

新手常犯的一个错误就是容易将子查询当成一个完全独立的数据集来使用。SQL 语言有一个与传统的编程语言很不一样的地方，就是 SQL 语言中并没有很强烈的“黑盒”概念。也就是说，编写一堆互相独立的子查询并把每个子查询当作一个“黑盒”数据块来看待，只要能得到最后结果就行，而不管其他，这种思路是错误的。它事实上割裂了子查询代码块内部处理逻辑与子查询代码块外部处理逻辑之间的联系，没有将整个 SQL 语句当成一个有机的整体来处理。从多个子查询中取数据与从多个表或者视图中取数据一样重要，代码写得不好效率就会很低。

示例 9-7 中演示了一个滥用子查询的例子，把子查询当黑盒使用的思想就会导致这种写法。

示例 9-7　滥用子查询

```
SELECT tract_id,
    (SELECT COUNT(*) FROM census.facts As F
WHERE F.tract_id = T.tract_id) As num_facts,
    (SELECT COUNT(*)
    FROM census.lu_fact_types As Y
    WHERE Y.fact_type_id IN (
        SELECT fact_type_id
        FROM census.facts F
        WHERE F.tract_id = T.tract_id
    )
    ) As num_fact_types
FROM census.lu_tracts As T;
```

上面的 SQL 语句如果改为示例 9-8 的写法效率会更高。下面的写法合并了多个 SELECT 动作并使用了关联查询机制，不但比上面的语句更简短，速度也更快。如果表的数据量很大或者硬件性能较差，这两种写法之间的性能差异会更明显。

示例 9-8　针对滥用子查询的语句的简化改写

```
SELECT T.tract_id,
    COUNT(f.fact_type_id) As num_facts,
    COUNT(DISTINCT fact_type_id) As num_fact_types
FROM census.lu_tracts As T LEFT JOIN census.facts As F ON T.tract_id = F.tract_id
GROUP BY T.tract_id;
```

图 9-4 显示的是示例 9-7 中的语句的执行计划，为了帮你解除查看字符型执行计划的痛苦，我们选择了以图形化方式展示。图 9-5 使用 http://explain.depesz.com 站点提供的工具将其执行计划转换为以 HTML 表格方式呈现，也可以提高你的查看效率。

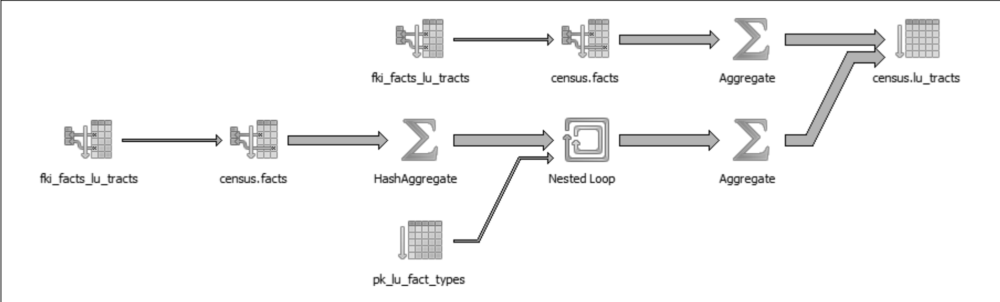


图 9-4：滥用子查询的 SQL 语句的执行计划图形化展示

HTML　TEXT　STATS

exclusive	inclusive	rows x	rows	loops	node
10.709	1292.135	↑ 1.0	1478	1	→ Seq Scan on lu_tracts t (cost=0.00..615535.37 rows=1478 width=12) (actual time
					SubPlan (forSeq Scan)
63.554	264.562	↑ 1.0	1	1478	→ Aggregate (cost=207.86..207.87 rows=1 width=0) (ac
153.712	201.008	↑ 1.0	68	1478	→ Bitmap Heap Scan on facts f (cost=4.79..207.69 rows=68 width=0) (actual time Recheck Cond: ((tract_id)::text = (t.tract_id)::text)
47.296	47.296	↑ 1.0	68	1478	→ Bitmap Index Scan on fki_facts_lu_tracts (cost=0.00..4.78 rows=68 width=0) (actual tim Index Cond: ((tract_id)::text = (t.tract_id)::text)
59.120	1016.864	↑ 1.0	1	1478	→ Aggregate (cost=208.56..208.57 rows=1 width=0) (ac
314.814	957.744	↑ 1.0	68	1478	→ Nested Loop (cost=207.86..208.39 rows=68 width=0) (actual ti
155.190	341.418	↓ 68.0	68	1478	→ HashAggregate (cost=207.86..207.87 rows=1 width=4) (actua
141.888	186.228	↑ 1.0	68	1478	→ Bitmap Heap Scan on facts f (cost=4.79..207.69 rows=68 width=4) (act Recheck Cond: ((tract_id)::text = (t.tract_ic
44.340	44.340	↑ 1.0	68	1478	→ Bitmap Index Scan on fki_facts_lu_trac (cost=0.00..4.78 rows=68 width=0) (ac Index Cond: ((tract_id)::text = (t.tract_i
301.512	301.512	↑ 1.0	1	100504	→ Index Scan using pk_lu_fact_types on lu_fact (cost=0.00..0.50 rows=1 width=4) (actual time Index Cond: (fact_type_id = f.fact_type_id)

图 9-5：滥用子查询的 SQL 语句的执行计划表格式展示

图 9-6 显示的是示例 9-8 中简化后语句的执行计划，从图中可以看到简化了多少执行步骤。

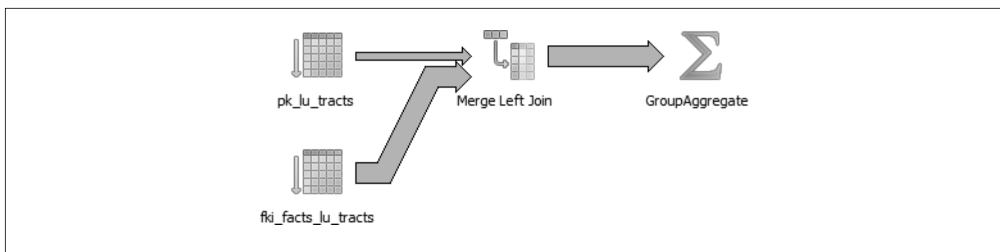


图 9-6：删除多余子查询后的执行计划图形化展示

请注意，我们并没有要求你完全不用子查询，只是建议你在确有必要时才使用，并且使用时应当注意考虑如何将子查询与 SQL 语句的主干融合，也许你会发现根本不需要通过子查询来实现你所需要的功能。总之请牢记：子查询不是独立的黑盒数据块，应与主语句通盘考虑后再结合使用。

9.3.2 尽量避免使用SELECT *语法

`SELECT *` 经常会导致性能浪费，会出现仅仅需要 10 页数据却查出 1000 页数据这种情况，这显然会导致网络传输负担加大，而且还会出现两个你可能意想不到的问题。

第一个问题与大对象有关。PostgreSQL 会使用 TOAST（The Oversized-Attribute Storage Technique，即超大尺寸属性存储技术）机制来存储二进制大对象以及超大文本。TOAST 机制会将超过主表存储限制的数据存储到一张辅助表中，并可能把单个文本字段拆分为多行存储。因此，读取超大字段就需要从辅助的 TOAST 表中查询出多条记录。举个例子，如果某表包含文本数据，其中存储了一整部《战争与和平》这么庞大的内容，然后你对这个表做了一个 `SELECT *` 操作，那么不难想象这个操作会慢到什么程度。

第二个问题与视图有关。我们定义视图的时候一般做不到完全精确地指定列，也就是说视图中一般会带若干可能不需要的列。PostgreSQL 的视图定义功能是很强大的，你可以使用 `SELECT *` 语句来定义视图，系统会自动将星号替换为目的表的完整字段列表，你也可以在视图定义语句中包含复杂运算表达式以及关联查询。这些创建视图的语句都是合法的，没有任何问题，但用户访问时就麻烦了，一旦对这种复杂视图执行 `SELECT *` 查询，那么视图定义中所有的复杂列都会经历漫长的运算过程，总体查询速度会很慢。

为了解释清楚以上观点，下面以示例 9-7 中带子查询的语句为基础创建一个视图，使用的基表是 census 中的表。

```
CREATE OR REPLACE VIEW vw_stats AS
SELECT tract_id,
    (SELECT COUNT(*)
      FROM census.facts As F
      WHERE F.tract_id = T.tract_id) As num_facts,
    (SELECT COUNT(*)
      FROM census.lu_fact_types As Y
      WHERE Y.fact_type_id IN (
```

```
            SELECT fact_type_id
            FROM census.facts F
            WHERE F.tract_id = T.tract_id
        )
        ) As num_fact_types
    FROM census.lu_tracts As T;
```

现在我们针对此视图执行以下语句：

```
    SELECT  tract_id FROM vw_stats;
```

在我们的测试环境中该语句的执行大约耗时 21 毫秒，速度很快，因为该语句没有访问 num_facts 和 num_fact_type 这两个视图字段，这两个字段需要经过复杂的运算才能得到结果。你查看一下该语句的执行结果就可以发现，其中没有任何一个步骤会访问 facts 表，因为规划器分析该语句后知道根本不需要访问此表。但如果我们使用下面的语句来访问上述视图：

```
    SELECT * FROM vw_stats;
```

在我们的环境中执行时间将飙升到 681 毫秒，其执行计划如图 9-4 所示。这里前后两个语句的性能差别是毫秒级，但如果表的记录数增加到千万级，列数增加到数百个，那么这两个语句的性能差异就非常大了。按照前一种写法，查询可以很快完成，你可以搞定后准点下班；按照后一种写法，你就得待在办公室加班来等这个查询执行完毕。

9.3.3　善用CASE语法

CASE 是 ANSI SQL 标准语法，其功能很强大，但利用它的人却很少，我们对此感到很惊讶。在很多需要聚合运算的场景中，使用 CASE 语法能够有效替代子查询。接下来我们使用两个例子来演示这一点，一个使用 CASE，一个使用子查询，然后比较二者的执行计划和性能差异。示例 9-9 使用了子查询语法。

示例 9-9　使用子查询而非 CASE

```
    SELECT T.tract_id, COUNT(*) As tot, type_1.tot AS type_1
    FROM
        census.lu_tracts AS T LEFT JOIN
        (SELECT tract_id, COUNT(*) As tot
            FROM census.facts
            WHERE fact_type_id = 131
            GROUP BY tract_id
        ) As type_1 ON T.tract_id = type_1.tract_id LEFT JOIN
        census.facts AS F ON T.tract_id = F.tract_id
    GROUP BY T.tract_id, type_1.tot;
```

图 9-7 是示例 9-9 的执行计划图示。

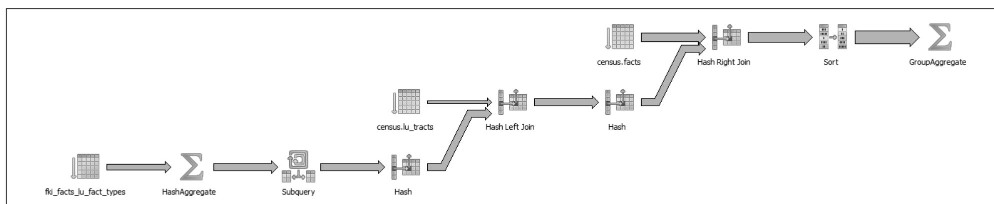


图 9-7：使用子查询而非 CASE 的执行计划

然后我们使用 CASE 语法改写这个查询。你会发现优化后的查询效率更高，也更容易理解，如示例 9-10 所示。

示例 9-10　使用 CASE 语法替代子查询

```
SELECT T.tract_id, COUNT(*) As tot,
    COUNT(CASE WHEN F.fact_type_id = 131 THEN 1 ELSE NULL END) AS type_1
FROM census.lu_tracts AS T LEFT JOIN census.facts AS F
ON  T.tract_id = F.tract_id
GROUP BY T.tract_id;
```

图 9-8 是示例 9-10 中语句的执行计划图示。

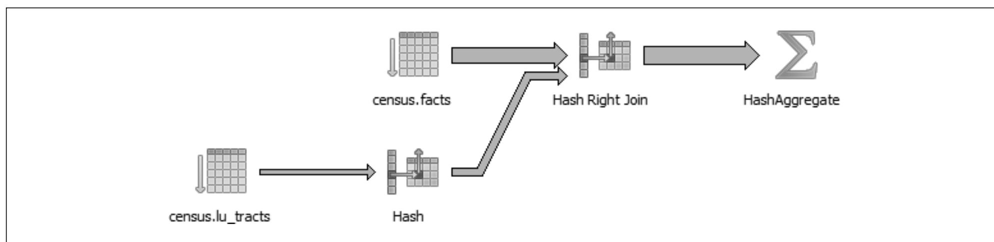


图 9-8：使用 CASE 代替子查询后的执行计划

尽管优化后的语句依然没用上 fact_type 索引，但其执行效率还是提升了，因为规划器仅对 facts 表做了一次扫描。一般来说，执行计划越短小，其执行过程就越容易理解，执行效率也越高，不过这并不是绝对的。

9.3.4　使用Filter语法替代CASE语法

PostgreSQL 9.4 中引入了新的 FILTER 关键字，7.15 节中介绍过其用法。在使用了 CASE 的聚合函数中总是可以用 FILTER 来代替 CASE，替换以后不但语法上看起来更整洁，而且在很多场景下执行效率也会有所提高。在示例 9-11 中，我们使用 FILTER 对示例 9-10 的语句进行了改写。

示例 9-11　使用 FILTER 语法来替代子查询

```
SELECT T.tract_id, COUNT(*) As tot,
    COUNT(*) FILTER (WHERE F.fact_type_id = 131) AS type_1
FROM census.lu_tracts AS T LEFT JOIN census.facts AS F
ON T.tract_id = F.tract_id
GROUP BY T.tract_id;
```

在我们的测试环境中，用 `FILTER` 替换 `CASE` 后，性能提升仅有大约 1 毫秒，而且两种写法的执行计划也基本类似。

9.4 并行化语句执行

并行化语句的执行过程会被规划器分发给多个后台进程去执行。通过这种并行执行方式，PostgreSQL 能够充分发挥多核处理器的威力，从而让语句执行更快完成。通过并行执行所能节省的时间根据服务器上 CPU 核数的多寡而有所不同，机器强大的情况下有可能是非常可观的。两核的机器上可能节省 50% 的时间，四核的机器上可能节省 75% 的时间。

并行化特性在 PostgreSQL 9.6 中引入，当时能够支持并行化的语句类型比较有限，一般仅包括那种比较直观的查询语句。但随着新版本的发布，我们希望有越来越多类型的语句能够支持并行化。

在 PostgreSQL 10 中，不能并行化的语句类型如下。

- 所有 DML 数据操纵语句，比如更新、插入和删除操作。
- 所有 DDL 数据定义语句，比如建表、加字段、建索引等。
- 在游标遍历过程中或者循环体中执行的查询语句。
- 部分聚合操作。常见的 `COUNT` 和 `SUM` 聚合都已支持并行化，但 `DISTINCT` 和 `ORDER BY` 还不支持。
- 自定义函数。默认情况下，自定义函数被设置为 `PARALLEL UNSAFE` 模式，即不能安全地进行并行化操作，但如果你确定你的自定义函数是可以并行的，那么可以通过 8.1 节中介绍的 `PARALLEL` 相关参数来启用自定义函数的并行化能力。

如需开启并行化语句执行能力，请遵循以下指导进行系统参数配置。

- `dynamic_shared_memory` 不允许设置为 `none`。
- `max_worker_processes` 需要设置为大于 0。
- `max_parallel_workers` 是 PostgreSQL 10 中新引入的一个参数，需要设置为大于 0 并且小于 `max_worker_processes` 的值。
- `max_parallel_workers_per_gather` 需要设置为大于 0 并且小于等于 `max_worker_processes` 的值。对 PostgreSQL 10 来说，该参数的值还必须小于等于 `max_parallel_workers`。该参数可以在会话级或者函数级进行设置。

9.4.1 并行化的执行计划是什么样子

我们如何知道一个语句已经真正地被并行化执行了呢？当然是通过执行计划查看最直观。并行化功能是通过规划器中一个名为采集节点（gather node）的模块实现的。因此如果你在执行计划中看到了 gather node 字样，那么说明该语句已经或多或少地用上了并行化功能了。一个采集节点仅包含一个执行计划，然后会把该执行计划分发给多个 worker 去执行。每个 worker 就是一个独立的 PostgreSQL 后台进程，每个后台进程负责处理整个查询中的一部分工作。所有 worker 的工作成果会被一个担任 leader 角色的 worker 搜集到一起。担任 leader 角色的 worker 和其他普通 worker 一样，也要处理 gather node 分派给自己的计算

任务，但它比其他普通 worker 多一项任务，就是要搜集所有普通 worker 的工作成果。如果 gather node 是整个执行计划的根节点，那么说明整个执行计划是彻底并行化执行的；如果 gather node 出现在比较低的层级中，那么只有对应层级的工作是并行化的。

为了便于调试，我们可以借助一个名为 `force_parallel_mode` 的参数。当该参数被设置为 `true` 时，规划器会用并行模式去执行所有可并行化的语句，即使规划器判定使用并行模式并不会比非并行模式快，它也会这么干。当需要判定一个语句为什么没有被并行化执行时，该参数非常有用。但请切记，不要在正式的生产环境中打开该参数！

目前为止，我们在本章中给出的例子都不会触发并行执行，因为将其并行化执行需要启动后台进程，而启动这些后台进程的成本已经超过了并行化带来的收益。为了确认有的语句在并行化执行的情况下反而比没有并行化更慢，先设置该参数：

```
Set force_parallel_mode = true;
```

然后再次运行示例 9-4，其输出结果如示例 9-12 所示。

示例 9-12　通过 `EXPLAIN (ANALYZE)` 命令得到并行执行计划

```
Gather
    (cost=1029.57..1051.65 rows=192 width=64)
    (actual time=12.881..13.947 rows=1 loops=1)
    Workers Planned: 1
    Workers Launched: 1
    Single Copy: true
    -> HashAggregate
        (cost=29.57..32.45 rows=192 width=64)
        (actual time=0.230..0.231 rows=1 loops=1)
        Group Key: "left"((tract_id)::text, 5)
        -> Bitmap Heap Scan on hisp_pop
            (cost=10.25..28.61 rows=192 width=36)
            (actual time=0.127..0.184 rows=204 loops=1)
            Recheck Cond:
                (((tract_id)::text >= '25025000000'::text) AND
                ((tract_id)::text <= '25025999999'::text))
                -> Bitmap Index Scan on hisp_pop_pkey
                    (cost=0.00..10.20 rows=192 width=0)
                    (actual time=0.106..0.106 rows=204 loops=1)
                    Index Cond:
                        (((tract_id)::text >= '25025000000'::text) AND
                        ((tract_id)::text <= '25025999999'::text))
Planning time: 0.416 ms
Execution time: 16.160 ms
```

可以看到，启动额外的 worker（即使只启动一个）带来的时间消耗显著地增加了总体查询所需的时间。

一般来说，对于耗时几毫秒的查询语句做并行化是没什么必要的。但需要访问极大数据集的查询通常需要耗时几秒甚至几分钟，此时并行化的价值就能体现出来了，初始化时启动额外 worker 的成本会远远小于并行化带来的收益。

为了演示并行化的威力，我们下载了美国劳动统计局的一份数据，其中包含 650 万条记录，然后对这些数据进行查询，如示例 9-13 所示。

示例 9-13　并行化的 GROUP BY 操作

```
set max_parallel_workers_per_gather=4;
EXPLAIN ANALYZE VERBOSE
SELECT COUNT(*), area_type_code
FROM labor
GROUP BY area_type_code
ORDER BY area_type_code;

Finalize GroupAggregate
    (cost=104596.49..104596.61 rows=3 width=10)
    (actual time=500.440..500.444 rows=3 loops=1)
    Output: COUNT(*), area_type_code
    Group Key: labor.area_type_code
    -> Sort
        (cost=104596.49..104596.52 rows=12 width=10)
        (actual time=500.433..500.435 rows=15 loops=1)
        Output: area_type_code, (PARTIAL COUNT(*))
        Sort Key: labor.area_type_code
        Sort Method: quicksort Memory: 25kB
        -> Gather
            (cost=104595.05..104596.28 rows=12 width=10)
            (actual time=500.159..500.382 rows=15 loops=1)
            Output: area_type_code, (PARTIAL COUNT(*))
            Workers Planned: 4
            Workers Launched: 4
            -> Partial HashAggregate
                (cost=103595.05..103595.08 rows=3 width=10)
                (actual time=483.081..483.082 rows=3 loops=5)
                Output: area_type_code, PARTIAL count(*)
                Group Key: labor.area_type_code
                Worker 0: actual time=476.705..476.706 rows=3 loops=1
                Worker 1: actual time=480.704..480.705 rows=3 loops=1
                Worker 2: actual time=480.598..480.599 rows=3 loops=1
                Worker 3: actual time=478.000..478.000 rows=3 loops=1
                -> Parallel Seq Scan on public.labor
                    (cost=0.00..95516.70 rows=1615670 width=2)
                    (actual time=1.550..282.833 rows=1292543 loops=5)
                    Output: area_type_code
                    Worker 0: actual time=0.078..282.698 rows=1278313 loops=1
                    Worker 1: actual time=3.497..282.068 rows=1338095 loops=1
                    Worker 2: actual time=3.378..281.273 rows=1232359 loops=1
                    Worker 3: actual time=0.761..278.013 rows=1318569 loops=1
Planning time: 0.060 ms
Execution time: 512.667 ms
```

为了拿未开启并行时的执行成本估算和执行时间来进行对比，我们设置 `max_parallel_workers_per_gather=0`，然后拿到执行计划，如示例 9-14 所示。

示例 9-14　非并行化执行的 GROUP BY 操作

```
set max_parallel_workers_per_gather=0;
EXPLAIN ANALYZE VERBOSE
SELECT COUNT(*), area_type_code
FROM labor
GROUP BY area_type_code
```

```
ORDER BY area_type_code;

Sort
    (cost=176300.24..176300.25 rows=3 width=10)
    (actual time=1647.060..1647.060 rows=3 loops=1)
    Output: (COUNT(*)), area_type_code
    Sort Key: labor.area_type_code
    Sort Method: quicksort  Memory: 25kB
    -> HashAggregate
        (cost=176300.19..176300.22 rows=3 width=10)
        (actual time=1647.025..1647.025 rows=3 loops=1)
        Output: count(*), area_type_code
        Group Key: labor.area_type_code
        -> Seq Scan on public.labor
            (cost=0.00..143986.79 rows=6462679 width=2)
            (actual time=0.076..620.563 rows=6462713 loops=1)
            Output: series_id, year, period, value, footnote_codes, area_type_code
Planning time: 0.054 ms
Execution time: 1647.115 ms
```

两种执行模式下得到的结果是一样的：

```
count   | area_type_code
--------+---------------
3718937 | M
2105205 | N
 638571 | S
(3 rows)
```

在并行化模式下，总共四个 worker，每个 worker 完成自己的工作大约耗时 280 毫秒。

9.4.2 并行化扫描

并行执行模式下有专门的扫描策略来将整个扫描任务切分给多个 worker。在 PostgreSQL 9.6 中，只有全表扫描可以并行执行。PostgreSQL 10 中新增了对位图堆扫描（bitmap heap scan）、索引扫描（index scan）以及仅索引扫描[1]（index-only scan）的并行化支持。然而对于索引扫描和仅索引扫描，目前还只有当使用了 B-树索引时才能实现并行。位图堆扫描则无此限制，基于任何类型索引的位图堆扫描操作过程都可以实现并行。但请注意，位图堆扫描过程中的建位图索引步骤是不可并行的，因此所有的 worker 必须等到位图索引构建完毕才能开始并行工作。[2]

注 1：指查询目标字段上全部都建有索引，因此仅需扫描索引即可得到结果，无须访问本表。——译者注

注 2：位图堆扫描过程简要说明如下：先根据表本体所占用的物理磁盘 PAGE 数建立一个位图区，其中每个 BIT 位都对应表本体上的一个物理磁盘 PAGE 页面。然后把从索引上查出来的目标记录位置写入该位图区，落在哪个 PAGE 页面就把其对应的 BIT 位置为 1。当整个位图索引构建完毕后，按照从前到后的顺序扫描表本体取数据，这样可以保证表本体上的每个 PAGE 只需读取一遍，相比不使用位图堆扫描技术的扫描方法大大节省了 IO。详情可参见 https://dba.stackexchange.com/questions/119386/understanding-bitmap-heap-scan-and-bitmap-index-scan。——译者注

9.4.3 并行化关联操作

关联操作也可以实现并行化。在 PostgreSQL 9.6 中，嵌套循环关联和散列关联这两种关联模式都是可并行化的。

在嵌套循环关联过程中，每个 worker 只需对分配给它的数据子集做关联运算，关联对象是所有 worker 共享的被关联表。

在散列关联过程中，每个 worker 会先构造一份全量的散列表，然后拿自己负责的那部分其他表数据和这个全量散列表做关联条件判定。由于每个 worker 都需要构建一个全量的散列表，而它自己不可能用到全量的散列数据，因此显然 worker 之间有很多冗余运算。所以，如果创建散列表是个很昂贵的操作，并行化的散列关联的速度会比非并行化关联操作慢。

PostgreSQL 10 中支持 merge 关联的并行化。merge 关联与散列关联存在类似的问题，即每个 worker 都要对关联操作某一侧的表进行完全处理，因此 worker 之间必然会存在重复运算。

9.5 人工干预规划器生成执行计划的过程

规划器生成执行计划的行为会受到多方面因素的影响，具体包括：是否有合适的索引、执行成本设置、执行策略设置以及数据如何分布等。本节将介绍多种可以对规划器施加人工干预的方法，通过这些方法可以得到更合理的执行计划。

9.5.1 策略设置

与某些其他数据库产品不同的是，PostgreSQL 查询规划器不接受索引提示，但你可以逐个查询或永久禁用各种策略设置（比如全表扫描、位图扫描、散列聚合、散列关联等都是一种执行策略，都有相应的策略设置），以阻止规划器选出某些效率较低的执行策略。PostgreSQL 官方手册中的“规划器方法配置”一节中介绍了所有规划器优化设置。默认情况下，所有策略设置都已启用，此时规划器因为不受什么约束所以灵活性是最大的。如果你对要查询的数据的特点预先有了一定了解，那么就可以有针对性地禁用某些策略来优化语句的执行路径。不过请注意，即使你设置了某种策略为禁用，也并不意味着规划器就一定不会使用该策略。规划器仅将这些设置当作用户的建议，最终的决定权还是在规划器。

我们有时候会将 `enable_nestloop`（嵌套循环）和 `enable_seqscan`（全表扫描）这两个设置设为“禁用”，因为这两种执行策略在大多数情况下的效率是很低的，当然并不是所有情况下都这样。你可以禁用这两种执行策略，但规划器在没有别的选择时还是可能会使用的，因为无论如何至少应该保证语句可以正常执行。如果你在执行计划中看到了全表扫描和嵌套循环这两种执行策略，那么建议核查一下到底是因为规划器已经找不到更好的策略所以不得不用，还是规划器选择了错误的策略。一个快速鉴别的办法是先禁用该策略，然后对禁用前后的执行计划进行比较。如果禁用前的执行计划中使用了该策略但禁用后却不使用了，那么再进一步比较一下这两种情况下执行计划的真实成本，就可以看出选择该策略到底是快了还是慢了。

9.5.2　你的索引被用到了吗

如果规划器选择了全表扫描策略，就意味着后续执行过程中会从头到尾读取表中的每一条记录。规划器会在以下两种情况下选择全表扫描策略：一种情况是表上没有合适的索引能满足查询条件的要求；另外一种是规划器认为通过索引来查找数据的成本要高于全表扫描。如果你禁用了全表扫描策略，但规划器依然选择了这么干，这就说明表上无索引或者虽然有索引但不适用此语句的查询条件。有两个错误是大家经常会犯的，一个是表上缺少必要的索引，另一个是索引建得不对而导致查询语句用不上。通过查询 `pg_stat_user_indexes` 和 `pg_stat_user_tables` 这两个视图可以很方便地得知你的索引是否被用上了。如需了解哪些语句执行得慢，请安装 `pg_stat_statements` 扩展包，其用法在 9.2 节中已经介绍过。

下面通过若干例子来说明。还是使用示例 7-22 中创建过的表，我们在该表的数组列 `fact_subcats` 上建立一个 GIN 索引，GIN 类型的索引是为数不多的能够支持数组类型的索引。语句如下：

```
    CREATE INDEX idx_lu_fact_types ON census.lu_fact_types USING gin  (fact_
subcats);
```

接下来执行一个查询语句以验证所建索引是否有效，查询条件为：`fact_subcats` 数组列中要包含“White alone”和“Asian alone”这两个元素。虽然全表扫描策略的默认设置就是“启用”，但我们还是显式地设定一下以防万一。示例 9-15 显示了该语句的执行计划。

示例 9-15　允许规划器选择全表扫描策略

```
set enable_seqscan = true;
EXPLAIN(ANALYZE)
SELECT *
FROM census.lu_fact_types
WHERE fact_subcats && '{White alone, Black alone}'::varchar[];

Seq Scan on lu_fact_types
    (cost=0.00..2.85 rows=2 width=200)
    (actual time=0.066..0.076 rows=2 loops=1)
         Filter: (fact_subcats
&& '{"White alone","Black alone"}'::character varying[])
        Rows Removed by Filter: 66
Planning time: 0.182 ms
Execution time: 0.108 ms
```

请注意，在启用全表扫描策略的情况下，规划器忽略了索引而选用了全表扫描策略。这可能是因为表的规模很小或者是因为索引不适用于本语句中的查询条件。在示例 9-16 中，我们执行的语句相同，但通过禁用全表扫描策略来强迫规划器使用了索引。

示例 9-16　禁用全表扫描策略，强行要求使用索引

```
set enable_seqscan = false;
EXPLAIN (ANALYZE)
SELECT *
FROM census.lu_fact_types
WHERE fact_subcats && '{White alone, Black alone}'::varchar[];
```

```
Bitmap Heap Scan on lu_fact_types
        (cost=12.02..14.04 rows=2 width=200)
    (actual time=0.058..0.058 rows=2 loops=1)
        Recheck Cond: (fact_subcats
&& '{"White alone","Black alone"}'::character varying[])
        Heap Blocks: exact=1
                -> Bitmap Index Scan on idx_lu_fact_types
        (cost=0.00..12.02 rows=2 width=0)
        (actual time=0.048..0.048 rows=2 loops=1)
        Index Cond: (fact_subcats
&& '{"White alone","Black alone"}'::character varying[])
Planning time: 0.230 ms
Execution time: 0.119 ms
```

通过该执行计划可以看到，索引建得没问题，是可以使用的，却使得整个查询的执行耗时更长，因为基于索引的查询成本要比基于全表扫描的成本高。因此，正常情况下规划器将选择使用全表扫描策略。但随着表中数据的增加，我们将会看到规划器优先选择索引查询策略。

为了与前面的例子做个对比，假设我们需要执行如下这样一个查询：

```
SELECT * FROM census.lu_fact_types WHERE 'White alone' = ANY(fact_subcats);
```

我们会看到，不管将 enable_seqscan 设为启用还是禁用，规划器总是会选择执行全表扫描，因为此时索引无法满足查询条件的需要。因此，建立合适的索引并编写正确的、能用上索引的 SQL 是很重要的。保证了这些以后，后续的工作就是多试验几次，以确保生成的执行计划是最优的。

9.5.3 表的统计信息

你可能会认为规划器神秘又强大，但无论如何规划器并不是神，它只是遵循一套设定好的算法来生成执行计划。关于规划器算法的内容细节远远超出了本书的范畴，在此不做讨论。虽然规划器的算法严重依赖于表的统计信息，但规划器不会在每次生成执行计划之前临时扫描所有的相关表以获取其统计信息，因为如果那么做的话任何语句的执行都将巨慢无比，完全没有执行效率可言，所以规划器会依赖预先搜集好的表统计信息。

因此，要想规划器能够做出准确的决定，及时准确地更新表统计信息是至关重要的。如果统计信息与实际情况相差太大，规划器很可能会常常推导出错误的执行计划，最差的情况就是错误地选择了全表扫描策略。一般来说，平均一张表只有 20% 的记录采样率，统计信息会基于这些参与采样的记录来生成。对于非常大的表来说，采样率可能更低。你可以通过设置 STATISTICS 值来修改在每一列上采样的行数。

通过查询 pg_stats 表，可以了解到规划器剔除了哪些统计信息以及使用了哪些统计信息，查询方法如示例 9-17 所示。

示例 9-17　数据分布直方图

```
SELECT
    attname As colname,
    n_distinct,
```

```
    most_common_vals AS common_vals,
    most_common_freqs As dist_freq
FROM pg_stats
WHERE tablename = 'facts'
ORDER BY schemaname, tablename, attname;

colname      | n_distinct | common_vals      | dist_freq
-------------+-----------+-----------------+----------------

fact_type_id |         68 | {135,113...      | {0.0157,0.0156333,...
perc         |        985 | {0.00,...        | {0.1845,0.0579333,0.056...
tract_id     |       1478 | {25025090300...  | {0.00116667,0.00106667,0.0...
val          |       3391 | {0.000,1.000,2...| {0.2116,0.0681333,0...
yr           |          2 | {2011,2010}      | {0.748933,0.251067}
```

pg_stats 表给出了表中指定列的值域分布图，规划器会根据此信息制订相应的执行计划。系统后台会有一个进程持续不断地更新 pg_stats 表。当表中插入或者删除大量数据后，你应该手动执行 VACUUM ANALYZE 来更新表的统计信息。VACCUM 指示将已删除的记录永久性地从表中移除，ANALYZE 指示更新表的统计信息。

对于经常参与关联查询并且在 WHERE 子句中频繁使用的列，应该考虑提升采样的行数。所需执行的代码如下：

```
ALTER TABLE census.facts ALTER COLUMN fact_type_id SET STATISTICS 1000;
```

PostgreSQL 10 中新增了对于多字段统计信息的支持，相应的语法是 CREATE STATISTICS。该特性使得用户能够针对多个字段的组合来创建统计对象。如果表中某些字段的值之间存在关联，那么多字段统计信息就能发挥作用。比如某些数据仅适用于某一特定年份，而不适用于其他年份。假设这两个字段分别为 fact_type_id 和 yr，那么可以针对这两个字段创建一个多字段统计信息，如示例 9-18 所示。

示例 9-18　多字段统计信息

```
CREATE STATISTICS census.stats_facts_type_yr_dep_dist (dependencies, ndistinct)
                ON fact_type_id, yr FROM census.facts;
ANALYZE census.facts;
```

CREATE STATISTICS 语句中必须指定同一张表中的两个或者多个字段。示例 9-18 中为 census.facts 表中的 fact_type_id 和 yr 字段创建了统计信息。另外，建议为统计信息起个名字，当然这个名字其实是可选的。如果你在起名时带了 schema 的名字（上例中 census.stats_facts_type_yr_dep_dist 里面的 census 就是 schema 的名字），那么该统计信息默认会被创建到该 schema 中，不指定则会被创建到默认 schema 中。

你可以采集两种统计信息，创建时必须至少指定其中一种。

- 第一种是数据依赖统计信息，记录了不同字段间值的依赖性。例如，只有当城市名为波士顿时才会出现邮政编码 02109。只有当优化器对带等值判定的语句进行优化时，数据依赖统计信息才能发挥作用，例如这种条件：city = 'Boston' and zip = '02109'。
- 第二种是 ndistinct 统计信息，记录了不同字段值一起出现的频率，并且会针对字段列表中每种排列组合都进行统计。ndistinct 统计信息专用于提升 GROUP BY 操作的效率，而且仅当 GROUP BY 子句中的字段全部落在统计信息所包含的字段列表中时，才能发挥作用。

CREATE STATISTICS 所创建的统计信息都存在 pg_statistic_ext 表中，可以使用 DROP STATISTICS 命令删除。与其他所有统计信息类似，该类统计信息是在 ANALYZE 命令执行期间计算出来的，而 ANALYZE 操作是由系统的自动清理及分析进程（官方名称为 autovacuum 进程）自动执行的。建议在建完表之后立即对其运行一次 ANALYZE 命令，这样该表的统计信息立即就可以被优化器用到。

9.5.4 磁盘页的随机访问成本以及磁盘驱动器的性能

另一个会影响规划器执行策略选择的设置是 random_page_cost（随机页访问成本比，简称 RPC）比率，它表示在磁盘上顺序读取和随机读取同一条记录的性能之比。一般来说，物理磁盘速度越快（一般也会越贵），该比率就会越小。RPC 的默认值是 4，该值适用于目前市面上的大多数机械硬盘。但如果使用的是固态硬盘或者 SAN 存储系统，有必要对此值进行调整。

你可以在 database、服务器、表空间这三个级别设置 RPC 比率。如果要在服务器级别设置该比率，直接在 postgresql.conf 文件中设置即可。如果同一台数据库服务器上使用了不同类型的硬盘，并且不同的表空间落在不同的硬盘上，那么可以在表空间级别设置 RPC 比率，语法如下所示。

```
ALTER TABLESPACE pg_default SET (random_page_cost=2);
```

有关该设置的详细信息，请参考“Random Page Cost Revisited”这篇博文。这篇文章建议采用以下设置。

- 高端 NAS/SAN 存储：2.5 或者 3.0
- 亚马逊 EBS 和 Heroku 云平台：2.0
- iSCSI 和其他普通 SAN 存储：6.0，但可能变化比较大，需要按照实际情况设定
- 固态硬盘：2.0 至 2.5
- NvRAM（也叫 NAND）：1.5

9.6 数据缓存机制

如果你之前执行过一个复杂且耗时较长的查询，那么后续再次执行此查询时会发现快了很多，这是因为系统的数据缓存机制发挥了作用。如果同一个查询语句按顺序多次执行，而且这些查询涉及的底层数据并没有发生变化，那么不管这些语句是被同一个用户还是多个用户执行，最终得到的结果都应该是一样的。只要内存中还有空间可用于缓存数据，那么规划器就可能会跳过生成执行计划和从磁盘读取表数据的步骤，直接从缓存中获取数据。如果语句中使用了 CTE 表达式和结果不变式函数（这类函数的运算结果不依赖外部数据，仅依赖输入的数据，也就是说固定的输入一定能得到固定的输出），那么系统会更加倾向于进行结果集缓存。

那么如何查看系统中缓存了哪些数据呢？可以通过安装 pg_buffercache 扩展包来查看。

```
CREATE EXTENSION pg_buffercache;
```

安装完毕后，你可以查询 pg_buffercache 视图，如示例 9-19 所示。

示例 9-19　查看表数据是否已被缓存

```
SELECT
    C.relname,
    COUNT(CASE WHEN B.isdirty THEN 1 ELSE NULL END) As dirty_buffers,
    COUNT(*) As num_buffers
FROM
    pg_class AS C INNER JOIN
    pg_buffercache B ON C.relfilenode = B.relfilenode INNER JOIN
    pg_database D ON B.reldatabase = D.oid AND D.datname = current_database()
WHERE C.relname IN ('facts','lu_fact_types')
GROUP BY C.relname;
```

示例 9-19 中的语句查出了 `facts` 和 `lu_fact_types` 这两张表中被缓存的页数。需要先实际执行一个 SQL 查询语句，才会有数据被真正缓存下来，此后示例 9-19 中的语句才能有结果。我们先执行一下下面这个语句。

```
SELECT T.fact_subcats[2], COUNT(*) As num_fact
FROM
        census.facts As F
        INNER JOIN
        census.lu_fact_types AS T ON F.fact_type_id = T.fact_type_id
GROUP BY T.fact_subcats[2];
```

当再次执行上述语句时，你应该可以看到至少 10% 的性能提升，并且示例 9-19 的查询可以看到类似以下的结果。

```
relname       | dirty_buffers | num_buffers
--------------+---------------+------------
facts         |             0 |         736
lu_fact_types |             0 |           4
```

用于缓存数据的内存大小是可指定的，该值越大，能缓存的数据就越多。postgresql.conf 中的 `shared_buffers` 就是用于设置此值的，但不应设得过大，否则会耗费过多时间去扫描缓存，反而降低了性能。

由于如今的物理内存已经极其廉价，因此一般不会再出现内存不够的情况。基于这一点，我们很容易就想到，可以将一些常用的表预先缓存到内存中，这样就可以提高后续访问效率。有一个名为 `pg_prewarm` 的扩展包可以用于实现此功能。`pg_prewarm` 会将指定的常用表预加载到缓存中，此后不管该表是首次被用户访问还是非首次访问，响应速度总是很快。

第10章 复制与外部数据

PostgreSQL 有很多方法可以实现与外部服务器或数据源之间的数据共享。第一种就是 PostgreSQL 自带的复制功能，通过该功能可以在另外一台服务器上创建出当前服务器的一个镜像。第二种方法是使用第三方插件，其中许多插件可以免费使用，并且其可靠性也是久经考验的。第三种方法是使用外部数据封装器（foreign data wrapper，FDW）。FDW 支持大量的外部数据源，从 9.3 版开始，有些 FDW 也开始支持对外部数据进行修改，包括 `postgres_fdw`、`hadoop_fdw` 和 `ogr_fdw`（详情请参见 10.3.4 节）。

10.1 复制功能概览

很多情况下我们都需要使用数据库复制功能，但不管具体场景如何，其根本原因都可以归结为两个：提升数据的可用性和可扩展性。确保可用性的手段是提供一台冗余的备用服务器，如果主服务器宕机了，备用服务器应立即接管并继续提供服务。对于规模较小的数据库来说，要达到此目标只需要保证你有另一台备用的物理服务器，并将数据库恢复到该服务器上即可。但对于规模很大的数据库（数据量为 TB 级别）来说，恢复过程本身可能需要好几个小时甚至几天，而且在此过程中系统无法对外提供服务。为尽量减少服务中断时间，你就需要使用复制功能。

我们需要复制功能的另一个原因是可扩展性诉求。假设你开设了一个网站，做着饲养并出售象鼩的买卖。做了几年之后，你已经拥有了几千只象鼩。全世界的客户都涌到你的网站来查看并购买，结果由于流量过大导致网站过载并无法正常处理请求，那么这时候就需要复制功能出马了。只需设定一个只读的从属服务器作为主服务器的镜像，然后就可以把海量的读请求分流到从属服务器上，只有当买家真正下单时才需要到主服务器上执行操作。

10.1.1 复制功能涉及的术语

在深入探讨复制功能之前，先介绍一下相关术语。

主服务器

主服务器是作为要复制数据的源头的数据库服务器，所有更新都在其上发生。使用 PostgreSQL 的内置复制功能时，仅允许使用一台主服务器。已有计划要支持多主服务器复制方案，请关注将来的版本。publisher（发布者）这个词你可能经常会看到，其含义是数据提供方。在 PostgreSQL 10 内置的逻辑复制功能中，publisher/subscriber（发布者 / 订阅者）这对术语会频频出现。

从属服务器

从属服务器使用复制的数据并提供主服务器的副本。尽管也有人谈到一些听起来更悦耳的术语（比如订阅者、代理等），但从属服务器这个名称仍是最贴切的。PostgreSQL 的内置复制功能目前仅支持只读从属服务器。

预写日志（write-ahead log，WAL）

WAL 就是记录所有已完成事务信息的日志文件，在其他数据库产品中一般称为事务日志。为了支持复制功能，PostgreSQL 将主服务器的 WAL 日志向从属服务器开放，然后从属服务器持续地将这些日志取到本地，然后将其中记载的事务重演一遍，这样就实现了数据同步。

同步复制

在事务提交阶段，PostgreSQL 需保证已经将此事务中所做的修改成功同步到 `synchronous_stand_names` 参数中所列出的至少一个从属服务器上，然后才能向用户反馈事务提交成功。在 PostgreSQL 9.6 之前，只需任何一个同步服务器反馈成功，事务就算成功。在 PostgreSQL 9.6 以及之后的版本中支持了一个增强功能，即可以要求至少 N 个从属服务器反馈成功后整个事务才算成功，这个 N 可以在 postgresql.conf 的 `synchronous_standby_names` 参数中进行配置。PostgreSQL 10 更进一步支持了 `FIRST` 和 `ANY` 这两个关键字，分别代表“只需第一个从属服务器反馈成功”和“只需任何一个从属服务器反馈成功”，这两个关键字也是在 `synchronous_standby_names` 参数中进行配置。如果不配置该项，则默认值为 `FIRST`，即只需第一个从属服务器反馈成功，PostgreSQL 9.6 在默认情况下的处理策略也是如此。

异步复制

在事务提交阶段，主服务器上提交成功就算成功，不需要等待从属服务器的数据更新成功。当从属服务器位于远端时该模式就比较有用了，因为可以避免网络延迟的影响。但有利必有弊，该模式下从属服务器的数据更新不够及时，与主服务器之间会有一些延迟。如果主从服务器之间的延迟很严重，以至于主从之间的差距大到主服务器上的 WAL 事务日志还没有传输到从属服务器就已经被清理掉，那么此时其实复制环境已经被破坏且无法恢复，从属服务器需要基于主服务器的数据重新初始化一遍。

为尽量降低出现上述问题的风险，PostgreSQL 9.4 中引入了复制槽（replication slot）的概念。所谓“复制槽”是指主从服务器之间的一种契约，该契约保证了在从属服务器消费到相应的 WAL 日志之前，这部分 WAL 日志不会被主服务器删除。这么做又会导致

另一个风险，就是如果某个持有“复制槽”的从属服务器发生故障或者与主服务器之间通信中断，那么主服务器上就不得不永久保留那些古老的 WAL 日志（因为不确定从属服务器是否已经消费这部分 WAL，所以不能删），从而导致磁盘空间被占满，最后导致服务器重启。

流式复制

流式复制模式下，WAL 日志并不是通过直接复制文件的方式从主服务器传递到从属服务器，而是通过基于 PostgreSQL 内部协议的消息来传递的。

级联复制

一个从属服务器可以把 WAL 日志传递给另一个从属服务器，而不需要所有的从属服务器都从主服务器取 WAL 日志，这进一步减轻了主服务器的负担。这种模式下，有的从属服务器可以作为同步的数据源，从而继续向别的从属服务器传播 WAL 数据，从这个角度看，其作用类似于主服务器。注意，这种扮演着“WAL 日志二传手”角色的从属服务器是只读的，它们也被称为**级联从属服务器**。

逻辑复制

这是 PostgreSQL 10 新支持的复制功能，通过该功能可以实现仅复制若干特定的表，而无须复制整个 PostgreSQL 服务器的所有数据，从而大大增加了复制的灵活性。该功能的实现依赖于一个名为**逻辑解码**（logic decoding）的机制，该机制可以将表数据的变更历史从 WAL 日志中以一种易于理解的格式解析出来，过程中无须关注数据库 WAL 日志格式的内部实现细节。逻辑解码机制从 PostgreSQL 9.4 开始就已支持，某些扩展包基于它实现了审计和复制功能。使用逻辑复制功能时，需要用到 `CREATE PUBLICATION` 和 `CREATE SUBSCRIPTION` 这两个 DDL 语法，前者表示哪些表要进行复制，后者表示哪些 PostgreSQL 服务器上的哪些 database 要订阅数据。

另外请注意，要使用该功能，需要把 `wal_level` 参数的值设置为 `logical`。

具体的使用示例可以参考“Logical Replication in PostgreSQL 10”这篇博客文章。

重新选主

重新选主是指从所有的从属服务器中选择一个并将其身份提升为主服务器的过程。PostgreSQL 9.3 引入了基于流消息复制的选主机制，该模式下选主时仅依靠流消息，而不再需要访问 WAL 日志文件，同时从属服务器也不需要经历一次重新复制过程。直到 9.4 版为止，重新选主还会要求整个数据库服务重启一次，将来的版本中可能会改进这一点。

PostgreSQL 的复制机制会复制所有事务性的变更，即在事务内发生的一切都可以被复制到从节点。PostgreSQL 中所有的 DDL 操作都是事务性的，因此建表、建视图、安装扩展包之类的操作都是可被复制的。注意非日志（unlogged）表上的插入、更新和删除操作是不记录 WAL 日志的，因此这些操作不会被复制。前面说过，安装扩展包的行为会被复制，所以当在主节点上安装扩展包时，要确保从节点上有扩展包的安装包，而且版本号需要和主节点一致，否则在主节点上执行 `CREATE EXTENSION` 操作会失败。

10.1.2 复制机制的演进

PostgreSQL 的复制机制本质上是靠 WAL 日志在主从节点间传递来实现的。流式复制要求从节点的操作系统以及 CPU 位数（32/64 位）必须和主节点相同。虽然官方并不强制要求主备节点上的 PostgreSQL 软件一直精确到第三位的小版本号都完全相同，但我们建议保证主备 PostgreSQL 的版本号完全相同。如果由于某些原因无法保证这一点，建议确保从节点的 PostgreSQL 小版本号高于主节点。

对复制机制的支持在以下 PostgreSQL 版本中不断演进。

- 9.4 版中新增了对复制槽的支持。所谓“复制槽”是指主从服务器之间的一种契约，该契约保证了在从属服务器尚未消费到相应的 WAL 日志之前，这部分 WAL 日志不会被主服务器删除。
- 9.5 版中引入了若干用于监控复制过程的函数,详情请参考官方手册中的“复制进度跟踪”部分。
- 9.6 版中支持了在同步模式下设置多个同步备节点的能力，目的是为了提升可靠性。
- PostgreSQL 10 中支持了原生的逻辑复制功能，该功能使得仅复制指定的若干张表成为可能。逻辑复制功能的另一个好处是可以让从节点有自己专属的表，这些表不参与复制，可以在从节点上对其进行任意修改。该版本还引入了“临时复制槽”的概念，通过它可以实现创建一次性的复制槽，一旦创建者会话退出，则该复制槽会自动消失。当通过 `pg_basebackup` 对整个服务器进行基础备份时，该功能特别有用。

  尽管逻辑复制是 PostgreSQL 10 才原生支持的新特性，但其实从 PostgreSQL 9.4 开始用户已经可以通过开源的 `pglogical` 扩展包来实现逻辑复制能力。如果你需要在不同的 PostgreSQL 大版本间进行复制，比如 PostgreSQL 10 和 PostgreSQL 9.4~9.6 之间，可以通过在两端的 PostgreSQL 服务器上都安装相应版本的 `pglogical` 扩展包来实现。如需在 PostgreSQL 10 和将来的新版本间进行逻辑复制，则无须借助 `pglogical`，直接使用原生的逻辑复制能力即可。

10.1.3 第三方复制解决方案

除了 PostgreSQL 自带的复制机制外，还有很多第三方提供的复制工具，Slony 和 Bucardo 是其中应用最广泛的两个，而且都是开源的。尽管 PostgreSQL 原生复制机制在每个版本中都会得到功能强化，但 Slony、Bucardo 以及其他第三方工具仍然在灵活性方面有着原生复制机制难以比拟的优势：它们支持仅复制单个 database 或者单张表；它们也不要求复制的源端和目的端的 PostgreSQL 版本和操作系统相同；它们还支持多主复制。但它们也有缺点：两个工具均依赖于新建额外的触发器来触发复制动作，同时还可能需要在被复制表上增加一些额外的字段，因此它们对系统架构有一定的侵入性；并且它们一般不支持建表、安装扩展包等 DDL 操作的同步。可以看出，这些第三方解决方案相比原生复制方案需要更多的人工干预，比如建触发器，为表加字段或者创建额外的视图，等等。

我们强烈建议你在决定使用哪一种产品之前参考一下“Replication, Clustering, and Connection Pooling”这篇文章。

10.2 复制环境的搭建

本节将介绍搭建用于针对整个 PostgreSQL 服务器的复制环境的所有步骤。我们将使用流式复制模式来实现，该模式基于主服务器和从属服务器之间的数据库连接来进行 WAL 日志传输。

10.2.1 主服务器的配置

主服务器的基本配置步骤如下所示。

(1) 创建一个专用于复制的用户账号。

```
CREATE  ROLE  pgrepuser REPLICATION LOGIN PASSWORD 'woohoo';
```

(2) 在 postgresql.conf 中设置好以下配置项。也有一种无须直接修改配置文件的方法：使用 `ALTER SYSTEM set 参数名=参数值`来修改配置项值，改好以后执行 `SELECT pg_reload_conf()` 来让配置生效。

```
listen_addresses = *
wal_level = hot_standby
archive_mode = on
max_wal_senders = 5
wal_keep_segments = 10
```

如果你希望基于逻辑复制来实现部分表的复制，那么要将 `wal_level` 设置为 `logical`。`logical` 模式相比 `hot_standby` 模式会记录更多事务日志，因此 `logical` 模式既适用于整个 PostgreSQL 服务器复制场景，又适用于指定部分表复制的场景。

以上设置在 PostgreSQL 官方手册的“服务器配置：复制”一节中有详细介绍。如果主从服务器的距离很远，而且主服务器上的事务处理又很繁忙，那么我们建议把 `wal_keep_segments` 参数设得大一点。如果运行的是 PostgreSQL 9.6 或更高版本，应该将 `wal_level` 设置为 `replica`，而不是上面写的 `hot_standby`。为了后向兼容，PostgreSQL 9.6 中会将 `hot_standby` 自动当成 `replica` 处理。

(3) 在 postgresql.conf 中添加 `archive_command` 配置指令，或者用 `ALTER SYSTEM` 修改，该参数的含义是 WAL 日志的保存路径。在流式复制模式下，该配置指令中的目标路径可设定为任何路径。更多有关此配置指令的信息，请参见 PostgreSQL 官方手册中的“PostgreSQL PGStandby 工具介绍”一节。

在 Linux/Unix 上，`archive_command` 行可参照如下格式设定：

```
archive_command = 'cp %p ../archive/%f'
```

也可以使用 `rsync` 命令替代 `cp` 以实现异地归档：

```
archive_command = 'rsync -av %p postgres@192.168.0.10:archive/%f'
```

在 Windows 上可按如下形式设定：

```
archive_command = 'copy %p ..\\archive\\%f'
```

(4) 在 pg_hba.conf 文件中设置一条权限规则，以允许从属服务器作为复制体系中的客户端连到主服务器。例如，以下这条规则所代表的含义是：允许你的私有网络中某台服务器上一个名为 pgrepuser 的 PostgreSQL 账号连接到主服务器，其 IP 地址范围为 192.168.0.1 到 192.168.0.254，验证方式为基于 MD5 的加密密码。

```
host replication pgrepuser 192.168.0.0/24 md5
```

(5) 重启 PostgreSQL 服务来让所有配置生效。

使用 PostgreSQL 安装路径下的 bin 文件夹中的 pg_basebackup 工具，来为整个 PostgreSQL 服务器创建一个全量备份。备份结果包含指定目录下的全量数据文件的副本。

使用 pg_basebackup 工具时，如果加上了 --xlog-method-stream 参数，则会把所有 WAL 日志也备份下来，过程中会建立一个数据库连接用于进行 WAL 复制；如果加上了 -R 参数，则会自动创建一个用于恢复的配置文件。



在 PostgreSQL 10 中，原先的 pg_xlog 目录已经被重命名为 pg_wal。

在下面的例子中，我们登录到从属服务器并针对主服务器（192.168.0.1）做了一次流式全量备份：

```
pg_basebackup -D /target_dir -h 192.168.0.1 \
--port=5432 --checkpoint=fast
--xlog-method=stream -R
```

如果你是为了实现备份的目的而使用 pg_basebackup，那么可以使用先打 TAR 包再做压缩这种常见输出形式，最终会在备份目标目录下为每个表空间生成一个 tar.gz 文件。注意，下面的命令行中的 -X 等同于 --xlog-method。由于流式日志不支持以压缩格式备份，所以需要把取日志的方式改为文件传递，命令行如下：

```
pg_basebackup -Z9 -D /target_dir/ -h 192.168.0.1 -Ft -Xfetch
```

除了备份全量数据外，我们一般还会将 WAL 日志也纳入备份体系中。为了实现这一点，在 PostgreSQL 10 之前，需要使用 pg_receivexlog 工具来获取事务日志，在 PostgreSQL 10 以及之后的版本中，该工具被改名为 pg_receivewal。只需为该工具设置一个定时任务，即可实现持续地导出并备份事务日志。

10.2.2 为从属服务器配置全量复制环境

如果搭建的是逻辑复制环境，请忽略本节内容。建议从属服务器与主服务器的各项系统配置完全相同，这会为你减少很多麻烦，特别是当你需要搭建一套用于保障系统高可用的主备倒换环境时，这一点尤其重要。此外，如果要在主服务器上安装扩展包，那么从节点上必须有相应扩展包的二进制安装文件，否则在主节点上执行 CREATE EXTENSION 后，从节点上执行恢复过程时就会报错。从属服务器节点必须能够处理主服务器发来的 WAL 事务日志。具体配置步骤如下。

(1) 新建一个数据库实例作为从属服务器，要求采用与主服务器相同的 PostgreSQL 版本（最好是小版本号也完全相同）。事实上，PostgreSQL 官方并不要求主服务器和从属服务器的 PostgreSQL 软件的小版本号也完全相同。其实你可以尝试一下，看看在不一样的情况下是否能够配置成功。
(2) 关闭从属服务器的 PostgreSQL 服务。
(3) 使用 pg_basebackup 导出的文件覆盖从属服务器上的相应文件。
(4) 将下面的配置设置添加到 postgresql.auto.conf 文件中。

```
hot_standby = on
max_connections = 20 #set to higher or equal to master
```

(5) 从属服务器的侦听端口不必与主服务器一样，因此可以选择在 postgresql.auto.conf 或者 postgresql.conf 中更改端口，也可以通过其他特定于操作系统的启动脚本进行更改，这些启动脚本会在启动之前设置 PGPORT 环境变量。
(6) 在 data 文件夹下创建一个名为 recovery.conf 的新文件，内容如下（注意下面第二行中要修改为真实的主机名、IP 地址和端口）。如果前面使用了 pg_basebackup 进行全量导出，那么该文件已自动生成，只需手动加上 trigger_file 那一行即可。

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.0.1 port=5432 user=pgrepuser ↵
password=woohoo application_name=replica1'
trigger_file = 'failover.now'
```

(7) 如果发现从属服务器处理事务日志的速度较慢，跟不上主服务器产生日志的速度，为避免主服务器产生积压，你可以在从属服务器上指定一个路径用于缓存暂未处理的日志。请在 recovery.conf 中添加如下一个代码行，该代码行在不同操作系统下会有所不同。
Linux/Unix 下：

```
restore_command = 'cp %p ../archive/%f'
```

Windows 下：

```
restore_command = 'copy %p ..\\archive\\%f'
```

本示例中，路径中指定的 archive 文件夹就是我们用于缓存日志的文件夹。

10.2.3 启动流复制进程

如果你已经用 pg_basebackup 完成了数据全量导出，那么请查看一下其中的 recovery.conf 文件的内容，确认是否均正常，然后再启动从属服务器。

此时所有主从属服务器应该都是能访问的。主服务器的任何修改，包括安装一个扩展包或者是新建表这种对系统元数据的修改，都会被同步到从属服务器。从属服务器可对外提供查询服务。

如果希望某个从属服务器脱离当前的主从复制环境，即此后以一台独立的 PostgreSQL 服务器身份而存在，请直接在其 data 文件夹下创建一个名为 failover.now 的空文件。从属服务器会在处理完当前接收到的最后一条事务日志后停止接收新的日志，然后将 recovery.conf 改名为 recovery.done。此时从属服务器已与主服务器彻底解除了复制关系，此后这台

PostgreSQL 服务器会作为一台独立的数据库服务器存在，其数据的初始状态就是它作为从属服务器时处理完最后一条事务日志后的状态。一旦从属服务器脱离了主从复制环境，就不可能再切换回主从复制状态了。要想切换回去，必须按照前述步骤一切从零开始。

10.2.4 使用逻辑复制实现部分表或者部分database的复制

PostgreSQL 10 支持通过逻辑复制能力实现仅复制部分表或者整个 PostgreSQL 服务器上的若干 database。逻辑复制的一大优势是支持在 PostgreSQL 10 和未来更新的版本间进行复制，而且不需要主从节点的操作系统和硬件架构相同。例如，我们可以实现一台 Linux 的 PostgreSQL 服务器与一台 Windows 的 PostgreSQL 服务器之间的复制。

在逻辑复制的概念体系中，数据提供方的服务器被称为“发布者”（publisher），数据接收方的服务器被称为“订阅者”（subscriber）。在发布者服务器上需复制的表所在的 database 中执行 `CREATE PUBLICATION`，即可将待复制的表作为数据源发布出去；在订阅者服务器上要接收数据的 database 中执行 `CREATE SUBSCRIPTION`，来指定要从哪个发布者服务器的哪个数据源来订阅数据。逻辑复制的主要问题在于它不支持 DDL 复制，因此要求待复制的表在开始复制之前在主从服务器都必须先建好。

假设我们的一台服务器上运行着两个 PostgreSQL 10 服务实例，发布端 PostgreSQL 的侦听端口是 5447，订阅端 PostgreSQL 的侦听端口是 5448。不管这两个 PostgreSQL 实例是运行在同一台服务器还是不同服务器上，以下过程都是一样的。在二者之间搭建逻辑复制的过程如下。

(1) 在发布端 PostgreSQL 实例上确保以下参数已配置：

```
SHOW wal_level
```

如果显示的值不是 `logical`，那么请使用以下命令修改：

```
ALTER SYSTEM SET wal_level = logical;
```

然后重启 PostgreSQL 服务。

如果需要实现级联复制，即订阅端服务器也需要作为发布者来向下一级订阅者进行发布，那么在订阅端服务器上最好也把 `wal_level` 配置成 `logical`。

(2) 在订阅端服务器上的目标 database 中创建待复制的表。如果你有许多表要复制或者要复制发布端某个 database 的所有表，那么建议你使用 `pg_dump` 将待复制的表结构导出来，然后拿到订阅端服务器上执行。例如，如果需要复制 postgresql_book 这个 database，通过以下语句可以导出其结构：

```
pg_dump -U postgres -p5447 -Fp --section pre-data --section post-data \
-f pub_struct.sql postgresql_book
```

然后用 psql 连接到订阅端服务器上执行前面导出的脚本，命令如下：

```
CREATE DATABASE book_sub;
\connect book_sub;
\i pub_struct.sql
```

(3) 然后在发布端服务器上针对待复制的 database 创建一个数据源。我们在这个例子中将使

用 CREATE PUBLICATION 来实现对某个 database 中所有表的复制。请注意，以下命令针对将来在此 database 中建立的表也生效，但要想这些后建的表能复制成功，需要先到订阅端服务器上把这些新表也创建一下：

```
CREATE PUBLICATION full_db_pub
    FOR ALL TABLES;
```

(4) 要想刚刚建立的发布数据源起作用，需要对其进行订阅消费。连接到订阅端服务器上的 book_sub 这个 database 中，然后执行以下命令：

```
\connect book_sub;
 CREATE SUBSCRIPTION book_sub
    CONNECTION 'host=localhost port=5447 dbname=postgresql_book \
user=postgres'
    PUBLICATION full_db_pub;
```

以上命令执行完毕后，查看一下订阅端 book_sub 库中的表，可以看到其中已经有了首次同步复制过来的数据。如果在发布端的 postgresql_book 库中插入新的记录，可以看到 book_sub 中已经复制过来了。

如果不再需要某个发布端或者订阅端，可以使用 DROP SUBSCRIPTION 和 DROP PUBLICATION 删除它们。

10.3 外部数据封装器

外部数据封装器（FDW）是 PostgreSQL 提供的一种用于访问外部数据源的手段，它是可扩展的，也兼容业界标准。该机制所支持的外部数据源包括 PostgreSQL 以及其他非 PostgreSQL 数据源。FDW 的核心概念是“外部表”，这种表看起来和当前 PostgreSQL 中其他表的用法完全相同，但事实上其数据本体是存在于外部数据源中的，该数据源甚至可能存在于另外一台物理服务器上。一旦定义好了外部表，其定义就会在当前数据库中持久化，你就可以放心地与使用普通表一样使用它，FDW 完全屏蔽了与外部数据源之间的复杂通信过程。比较流行的 FDW 及其用法示例可以通过 PostgreSQL FDW 维基百科页面查到。可以通过 PGXN FDW 页面和 PGXN Foreign Data Wrapper 页面查到 PostgreSQL 的 FDW 目录。在 GitHub 上搜索“PostgreSQL Foreign Data Wrappers”，可以搜索到前述很多 FDW 的源码，另外还能找到一些不在前述列表中的 FDW。如果你需要自行封装某个外部数据源，那么请先到前面提供的这些站点中查询一下别人是否已实现，如果没有，再自己做。如果封装成功，请记得发布出来与社区分享。

大多数 PostgreSQL 安装包会默认携带两个 FDW：file_fdw 和 postgres_fdw，但默认没有安装，你可以执行 CREATE EXTENSION 来安装它们。

直到 PostgreSQL 9.2 为止，FDW 仅支持对外部表进行查询。PostgreSQL 9.3 中引入了一组新的 API，实现了对外部表的修改。但 PostgreSQL 自带的 FDW 中只有 postgres_fdw 支持此特性。

本节中，我们将向你演示如何注册外部服务器、外部用户以及外部表，最后介绍如何查询外部表。我们使用的例子中都使用 SQL 命令行来创建和删除对象，你也可以通过 pgAdmin 的图形化界面工具来实现相同的操作。

10.3.1　查询平面文件

可以使用 `file_fdw` 这个 FDW 来查询平面文件，它是以扩展包的形式存在的，因此可以通过以下 SQL 安装：

```
CREATE EXTENSION file_fdw;
```

尽管通过 `file_fdw` 可以直接读取数据库实例所在的本地服务器上的文件，但为了和别的 FDW 在语义上保持一致，还是得定义一个逻辑上的外部服务器。请执行以下命令来定义一个“伪”外部服务器：

```
CREATE SERVER my_server FOREIGN DATA WRAPPER file_fdw;
```

接下来要注册外部表。你可以将外部表置于任何一个 schema 中，但我们一般是创建一个单独的 schema 来容纳所有的外部表。接下来将使用一个名为 `staging` 的 schema，如示例 10-1 所示。

这里先列出几行我们将要访问的外部文件的内容，每一行内的字段用管道符分隔，格式如下：

```
Dev|Company
Tom Lane|Crunchy Data
Bruce Momjian|EnterpriseDB
```

示例 10-1　定义基于分隔符格式文件的外部表

```
CREATE FOREIGN TABLE staging.devs (developer VARCHAR(150), company VARCHAR(150))
SERVER my_server
OPTIONS (
    format 'csv',
    header 'true',
    filename '/postgresql_book/ch10/devs.psv',
    delimiter'|',
    null''
);
```

上面的示例中，尽管外部表映射到一个用管道符作为分隔符的平面文件，但我们依然将其标识为“csv”格式。有人可能会根据 CSV（comma seperated values）的名称认为只有用逗号作为分隔符的文件才能称为 CSV，但在 FDW 的术语体系中，CSV 文件就是以某种分隔符来区分列值的平面文件，不管这个分隔符具体是什么字符，都可以称之为 CSV。

上述定义步骤完成后，你就可以直接通过 SQL 访问外部表了：

```
SELECT * FROM staging.devs WHERE developer LIKE 'T%';
```

如果不再需要此外部表，可以删掉：

```
DROP FOREIGN TABLE staging.devs;
```

10.3.2　以不规则数组的形式查询不规范的平面文件

通常，平面文件在每一行中会有许多不同的列，并且包含多个标题行和页脚行。我们最喜欢使用 `file_textarray_fdw` 这个 FDW 来处理这种非结构化平面文件。该 FDW 能处理任何带分隔符的平面文件，即使每一行中的元素数量不一致也没问题，因为它可以将每一行

作为一个变长的文本数组（`text[]`）来进行处理。

问题是 `file_textarray_fdw` 不是 PostgreSQL 原生支持的扩展包，因此你必须手动编译安装它。首先，在安装 PostgreSQL 时需要附带安装系统头文件，以便于后续的编译。然后从 Adunstan GitHub 这个站点下载 `file_textarray_fdw` 的源码。请注意：该站点为每个 PostgreSQL 版本都准备了相应的源码，请确保选择的是正确的版本。在编译完成后，请将它以扩展包的形式安装好，该过程与前面介绍过的安装其他 FDW 的过程完全相同。

如果你的环境是 Linux/Unix，只要安装了 `postgresql-dev` 这个包，那么编译过程是很简单的，但在 Windows 上就比较麻烦，所以我们已经为你编译好了安装包，你可以从以下几个链接中选择合适的版本下载。Windows 32/64 位适配 PostgreSQL 9.4：http://www.postgresonline.com/journal/archives/340-Foreign-Data-Wrappers-for-PostgreSQL-9.4-Windows.htmlhttp:/bit.ly/2oRDY6X。Windows 32/64 位适配 PostgreSQL 9.5 和 PostgreSQL 9.6：http://www.postgresonline.com/journal/archives/361-Foreign-Data-Wrappers-for-PostgreSQL-9.5-windows.html。

FDW 安装好以后，首先创建扩展包。

```
CREATE EXTENSION file_textarray_fdw;
```

然后就跟安装其他 FDW 时一样，创建好外部服务器。

```
CREATE SERVER file_taserver FOREIGN DATA WRAPPER file_textarray_fdw;
```

然后设置外部表。可以将外部表放入任何一个你认为合适的 schema 中。在示例 10-2 中，我们将再次使用前面用过的 `staging` schema。

示例 10-2　创建一个基于文本数组的外部表

```
CREATE FOREIGN TABLE staging.factfinder_array (x text[])
SERVER file_taserver
OPTIONS (
    format 'csv',
    filename '/postgresql_book/ch10/DEC_10_SF1_QTH1_with_ann.csv',
    header 'false',
    delimiter ',',
    quote '"',
    encoding 'latin1',
    null ''
);
```

假设我们要处理这样一个 CSV 文件：文件中含有 8 个标题行，而列数多得我们不想数。当前述设置步骤都完成后，就可以直接查询这个文件的内容了。通过以下查询可以得到标题行的名称，这些标题行的第一个列标头为 `GEO.id`。

```
SELECT unnest(x) FROM staging.factfinder_array WHERE x[1] = 'GEO.id'
```

以下查询能查出数据的前两列。

```
SELECT x[1] As geo_id, x[2] As tract_id
FROM staging.factfinder_array WHERE x[1] ~ '[0-9]+';
```

10.3.3　查询其他PostgreSQL服务实例上的数据

从 9.3 版开始，大多数的 PostgreSQL 发行包都包含了 postgres_fdw 这个 FDW。通过它还可以对其他 PostgreSQL 服务器上的数据进行读取和修改操作，哪怕两边的 PostgreSQL 版本不一致也没关系。

首先也是要进行 FDW 扩展包的安装。

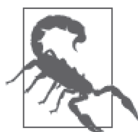
```
CREATE EXTENSION postgres_fdw;
```

然后创建外部服务器。

```
CREATE SERVER book_server
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', port '5432', dbname 'postgresql_book');
```

如果创建好外部服务器后需要更改连接选项或将其添加到外部服务器，则可以使用 ALTER SERVER 命令。比如，如果需要更改你所指向的服务器，可以执行以下代码行。

```
ALTER SERVER book_server OPTIONS (SET host 'prod');
```



对于主机、端口和 database 这几项的连接设置的更改只对新建立的会话生效，不会影响已有的会话。原因是会话在开始时建立，此后就一直复用，而不会断开重连。

然后创建一个用户映射关系[1]，将远端的某个角色映射到本地的 public 角色。

```
CREATE USER MAPPING FOR public SERVER book_server
OPTIONS (user 'role_on_foreign', password 'your_password');
```

注意，上述映射关系中的远端角色必须是一个已存在的角色，并且有登录权限。这样任何能连到本地数据库的用户都可以连到远端数据库。

现在可以创建外部表了。该表可以映射远端表的全部或部分列。在示例 10-3 中，我们创建了一个外部表，映射到远端数据库上的 censu.facts 表。

示例 10-3　定义一个映射到远端 PostgreSQL 数据库的外部表

```
CREATE FOREIGN TABLE ft_facts (
    fact_type_id int NOT NULL,
    tract_id varchar(11),
    yr int, val numeric(12,3),
    perc numeric(6,2)
)
SERVER book_server OPTIONS (schema_name 'census', table_name 'facts');
```

上面的示例仅包含外部表的最基本的选项。默认情况下，映射到远端 PostgreSQL 数据库的外部表都是可更新的，当然前提是映射关系中所使用的远端数据库角色对映射的远端表要有修改权限。该 updatable 设置是一个布尔设置，可以在定义外部表或者外部服务器时进

注 1：所谓“用户映射”是指在远端服务器的某个角色和本地服务器的某个角色之间建立对应关系，这样本地角色可以用远端角色的权限来操作远端服务器上的数据。——译者注

行更改。例如，如果想把外部表设定为只读，可以执行以下代码行。

```
ALTER FOREIGN TABLE ft_facts OPTIONS (ADD updatable 'false');
```

要将表设置回 `updatable` 状态，请执行以下代码行。

```
ALTER FOREIGN TABLE ft_facts OPTIONS (SET updatable 'true');
```

表级别上的 `updatable` 属性会替代外部服务器设置。

`ALTER FOREIGN TABLE` 语句除了可以更改 `OPTIONS` 之外，还可以添加或者删除列，语法是 `ALTER FOREIGN TABLE .. DROP COLUMN`。

PostgreSQL 9.5 中引入了对 `IMPORT FOREIGN SCHEMA` 命令的支持，它可以实现外部表的自动创建，从而为用户节省大量时间。不过请注意，并不是所有的 FDW 都支持 `IMPORT FOREIGN SCHEMA` 能力。每个 FDW 在执行外部表结构导入时都可以设置一些服务端参数。对于 `postgres_fdw` 来说，支持的参数如下。

`import_collate`

是否将远端 PostgreSQL 服务器上的字符排序规则设置也导入到本地外部表。默认为 `true`。

`import_default`

是否将远端 PostgreSQL 服务器上的字段默认值属性也导入到本地外部表中。默认为 `false`，即本地服务器上的外表字段上没有默认值。当要对外表进行数据插入操作时，默认值是有用的：如果 `Insert` 语句中未包括某字段，则插入结果中会取该字段的默认值，因此理论上应该要把默认值信息复制过来。但有一点请特别注意：如果该字段的默认值是基于一个序列号生成器的自动递增值，则其效果未必符合你的预期，因为本地服务器上得到的序列号和远端服务器上得到的序列号完全可能不一样。

`import_not_null`

是否将远端 PostgreSQL 服务器上的字段 NOT NULL 属性导入到本地外部表中。默认为 `true`。

在示例 10-4 中，我们将远端 PostgreSQL 服务器上 books.public 这个 schema 中的所有表结构导入本地服务器并自动生成了外部表。

示例 10-4　使用 `IMPORT FOREIGN SCHEMA` 命令来导入一个 schema 中的所有表结构

```
CREATE SCHEMA remote_census;
IMPORT FOREIGN SCHEMA public
FROM SERVER book_server
INTO remote_census
OPTIONS (import_default 'true');
```

如上例 10-4 所示，`IMPORT FOREIGN SCHEMA` 命令会为外部 schema 中的每张表在本地的 `remote_census` schema 中创建一张同名的外部表。

如果只想导入部分表，可以使用 `LIMIT TO` 或者 `EXCEPT` 子句。例如，如果只希望导入 `facts` 和 `lu_fact_type` 这两张表，可以这么写：

```
IMPORT FOREIGN SCHEMA census
    LIMIT TO (facts, lu_fact_types)
    FROM SERVER book_server INTO remote_census;
```

如果 LIMIT TO 后面指定的表在远端 PostgreSQL 服务器上不存在，系统会直接忽略掉，不会报错。我们建议在执行完导入外部表操作后进行一次检查，以确保所有你希望导入的表的确均已导入成功。

EXCEPT 与 LIMIT TO 用法类似，只不过二者效果正好相反：EXCEPT 用于指定哪些表不要导入；LIMIT TO 用于指定哪些表要导入。

如果你在 PostgreSQL 系统中使用了一些扩展包提供的能力，那么可能需要用到外部服务器的一个新参数，该参数名为 extensions，是 PostgreSQL 9.6 中引入的，它可以用来提升外部服务器的访问性能。要想用上此特性，请按如下语法为现有的 postgres_fdw 外部服务器进行参数设置：

```
ALTER SERVER census(OPTION ADD extensions 'btree_gist, pg_trgm');
```

这个 extensions 参数的内容是一个用逗号分隔的扩展包列表，它表示远端 PostgreSQL 服务器上已经安装了哪些扩展包。当 PostgreSQL 要执行的语句的 WHERE 条件中涉及扩展包中定义的数据类型或者函数时，系统会尝试将这些函数调用推送到远端服务器上去执行，这样性能就得以提升。如果未设置 extensions 参数，所有扩展包中的函数都不得不在本地服务器上执行，这就意味着需要先把所有相关的数据从远端服务器上传输到本地，从而大大地增加了要通过网络传输的数据量，速度自然也会受影响。

10.3.4 使用ogr_fdw查询其他二维表形式的数据源

有许多 FDW 可以用来查询其他关系型数据库或者平面文件数据，其中大多数都仅支持某种特定类型的数据源。例如，有专门用于查询 MongoDB 数据的 MongoDB FDW，有专门用于查询 Hadoop 数据源的 Hadoop FDW，也有专门用于查询 MySQL 数据源的 MySQL FDW。

我们知道有两种 FDW 能够支持多种数据源。第一个是 Multicorn FDW，它事实上是一种支持用户以 Python 语言编写自定义 FDW 的 FDW API 平台。已经有一些基于 Multicorn FDW 平台的 FDW 可用，但该平台不支持 Windows 而且在 Linux 上用起来也问题很多，要想用好需要一些技巧。

另一个支持多种数据源的 FDW 就是 ogr_fdw，本节将详细展示它的用法。ogr_fdw 支持很多二维表形式的数据源，例如电子表格（比如 Excel 或者 LibreOffice 等）软件使用的表单、Dbase 文件、CSV 文件以及其他关系型数据库等。另外，它还支持空间数据，可以从 SQL Server、Oracle 等关系型数据库把数据导入到 PostGIS 并转换为 PostgreSQL 几何类型。

有些 PostGIS 的安装包中同时也会附带提供 ogr_fdw 扩展包。例如，EnterpriseDB 公司的 StackBuilder 安装工具中提供的 Windows 版 PostGIS 包里面就附带提供了 ogr_fdw；在 CentOS 和红帽 Linux 企业版（RHEL）操作系统中，使用 yum 一站式安装工具从 yum.postgresql.org 站点也能获取到 ogr_fdw；在 BigSQL 公司提供的 PostgreSQL 发行版安装包中也提供了 ogr_fdw。如果你希望根据自行编译安装 ogr_fdw，可以从 GitHub 上下载到源码。

从内部实现机制来看，ogr_fdw 其实是利用了“地理空间数据抽象层库”（Geospatial Data

Abstraction Library，GDAL）来实现上述强大功能。因此，在编译或者使用 `ogr_fdw` 之前，需要先编译并安装好 GDAL 库。GDAL 库历史悠久、功能特性繁多，因此根据不同的编译选项可以编译出多种多样的能力组合。因此请注意：你的 GDAL 库和我的 GDAL 库有可能差别很大。GDAL 一般是作为 PostGIS 的一部分随之安装，所以为了确保在使用 GDAL 的过程中不会遇到什么问题，我们建议安装最新版本的 PostGIS。

GDAL 库安装好以后，一般都自带对 Excel 表格、LibreOffice Calc 表格、ODBC 数据源、空间数据 Web 服务数据源的支持。在 Windows 上它一般还会支持 Microsoft Access 数据源，但在 Linux/Mac 下一般不会支持。

安装好 `ogr_fdw` 的二进制安装包以后，连到 PostgreSQL 上要安装 `ogr_fdw` 的 database，然后执行以下命令即可：

```
CREATE EXTENSION ogr_fdw;
```

由于 `ogr_fdw` 支持多种多样的外部数据源，因此针对不同的数据源来说，外部服务器的含义也是不一样的。比如对于 CSV 文件来说，CSV 文件所在的目录就对应于外部服务器，该目录下的每个 CSV 文件对应于一张独立的外表；对于 Microsoft Excel 和 LibreOffice Calc 来说，一张工作表就对应于一个外部服务器，这张工作表中的每一张表单就对应于一个独立的外表；对于 SQLite 数据库来说，一个 database 就对应于一个外部服务器，其中的每一张表就对应一个独立的外表。

在下面的例子中，我们把一个 LibreOffice 工作表映射为一个外部服务器，把其中的表单映射为独立的外表：

```
CREATE SERVER ogr_fdw_wb
FOREIGN DATA WRAPPER ogr_fdw
OPTIONS (
    datasource '/fdw_data/Budget2015.ods',
    format 'ODS'
);

CREATE SCHEMA wb_data;
IMPORT FOREIGN SCHEMA ogr_all
FROM SERVER ogr_fdw_wb INTO wb_data;
```

上面的 `ogr_all` 是一个用来泛指所有 schema 的代称而非真实的 schema 名，它代指了数据源中所有的 schema，其作用是将数据源中所有的表全部导入，不管表属于哪个 schema。值得注意的一点是，有的外部数据源有 schema 概念而有的数据源没有。为了适应各种数据源，当数据源不提供 schema 概念时，`ogr_fdw` 可以接受一个虚拟的 schema 名（填在上例中 `ogr_all` 出现的位置），这个虚拟 schema 名其实是表名前缀，也就是说，所有名字符合此前缀的表都被认为属于该虚拟的 schema，从而可以被一次性导入。例如，如果希望一次性把所有名字以“Finance”开头的表单导入进来，那么可以把上面脚本中的 `ogr_all` 替换为“Finance”：

```
CREATE SCHEMA wb_data;
IMPORT FOREIGN SCHEMA "Finance"
FROM SERVER ogr_fdw_wb INTO wb_data;
```

schema 的名字是区分大小写的，因此如果表单的名字中含有大写字符或者非标准字符，需

要在其前后加引号。

下一个例子是对一个存有 CSV 文件的目录创建外部数据源服务器。创建一个名为 ff 的 schema 来容纳外部表。ogr_fdw 会在 ff 这个 schema 中自动针对每个名字以 Housing 开头的 CSV 文件创建外表。脚本如下：

```
CREATE SERVER ogr_fdw_ff
FOREIGN DATA WRAPPER ogr_fdw
OPTIONS (datasource '/fdw_data/factfinder', format 'CSV');
CREATE SCHEMA ff;
IMPORT FOREIGN SCHEMA "Housing"
FROM SERVER ogr_fdw_ff INTO ff;
```

假设在上例的目录中有 Housing_2015.csv 和 Housing_2016.csv 这两个文件，那么系统会为它们在 schema ff 中各建一个外表，表名为 housing_2015 和 housing_2016。

ogr_fdw 默认会对外部数据源中的表名和字段名进行转换：所有大写的表名和字段名都会被修改为小写。如果你不希望发生这种转换，可以在 IMPORT FOREIGN SCHEMA 命令中加上一些参数，来使得外部数据源的表名和字段名维持原样不变。例如：

```
IMPORT FOREIGN SCHEMA "Housing"
    FROM SERVER ogr_fdw_ff INTO ff
     OPTIONS(launder_table_names 'false', launder_column_names 'false');
```

这个语句创建出来的外表名就是 Housing_2015 和 Housing_2016，与外部数据源中的原始表名保持一致。

10.3.5　查询非传统数据源

长久以来，数据库界多元化的趋势有增无减，各种架构大相径庭的数据库如雨后春笋般层出不穷，要想紧跟业界潮流十分困难。这些异彩纷呈的数据库中，有的只是昙花一现，喧闹一阵子就消失无踪，有的立志要将传统的关系型数据库挑落马下，更有另类者甚至看起来根本就不像是数据库。FDW 功能的引入就是 PostgreSQL 对这一百花齐放局面的应对策略之一。不管外界如何风云变幻，PostgreSQL 无须改变自身的核心功能，而是通过 FDW 搭建与这些异构数据库之间沟通的桥梁。

在下面的例子中，我们将展示如何使用 www_fdw 来查询来自 Web 服务的数据。该示例是从 www_fdw Examples 站点借鉴而来的。

PostgreSQL 发行包是不附带 www_fdw 的，因此需要自行编译安装。如果你使用的是 Linux/Unix 环境并且已安装了 postsql-dev 这个包，那么编译是很容易的。请从 https://github.com/cyga/www_fdw 下载最新版本的 www_fdw 源码。对于 Windows 平台来说，我们已经替你编译好了，下载链接如下：

Windows-32 9.1（http://www.postgresonline.com/downloads/fdw_win32_91_bin.zip）

Windows-64 9.3（http://www.postgresonline.com/downloads/fdw_win64_93_bin.zip）

首先安装 FDW 扩展包。

```
CREATE EXTENSION www_fdw;
```

然后创建针对 Google Web 服务的外部服务器。

```
CREATE SERVER www_fdw_server_google_search
    FOREIGN DATA WRAPPER www_fdw
    OPTIONS(uri 'http://ajax.googleapis.com/ajax/services/search/web?v=1.0');
```

`www_fdw` 默认支持 JSON 格式数据源，因此我们不需要在上述语句的 `OPTIONS` 修饰符中特地声明数据源格式。此外 `www_fdw` 还支持 XML 格式数据源。如果想了解 `www_fdw` 所支持的形参的详细信息，请参阅其官方文档 `www_fdw`。请注意：每一个 FDW 都有其独特的设置，互不相同。

接下来选定至少一个本地角色来创建 FDW 用户映射关系。每个能连到本地库上的用户都应该有权限访问 Google 搜索服务器，因此我们将远端数据源的访问权限映射给本地的 `public` 角色。

```
CREATE USER MAPPING FOR public SERVER www_fdw_server_google_search;
```

然后创建外部表，如示例 10-5 所示。表中每个字段对应于 Google 自动生成的 URL 搜索路径中的一个 GET 参数。

示例 10-5　基于 Google Web 服务数据源创建一个外部表

```
CREATE FOREIGN TABLE www_fdw_google_search (
    q text,
    GsearchResultClass text,
    unescapedUrl text,
    url text,
    visibleUrl text,
    cacheUrl text,
    title text,
    content text
) SERVER www_fdw_server_google_search;
```

前面设置用户映射关系时未指定任何权限，因此需要进行一次授权动作，然后才可以访问外部表。

```
GRANT SELECT ON TABLE www_fdw_google_search TO public;
```

请注意，现在最精彩的部分来了：我们以 `New in PostgreSQL 10` 作为关键词进行搜索，并用正则表达式筛选掉返回结果中的 HTML 标签，语句如下所示。

```
SELECT regexp_replace(title,E'(?x)(< [^>]*? >)','','g') As title
FROM www_fdw_google_search
WHERE q= 'New in PostgreSQL 10'
LIMIT 2;
```

瞧吧！我们真的得到了想要的搜索结果。

```
title
---------------------
PostgreSQL 10 Roadmap
PostgreSQL: Roadmap
(2 rows)
```

附录A

PostgreSQL的安装

A.1 Windows以及桌面Linux环境

EnterpriseDB 公司为 Windows 和桌面 Linux 环境提供了安装包，而且针对每种 OS 都同时提供了 32 位和 64 位的包。

基于图形化界面的安装包使用起来非常简单，其中还附带了 pgAdmin 以及一款辅助安装工具 StackBuilder。通过 StackBuilder，可以为 PostgreSQL 安装一些插件和辅助工具，比如 JDBC 驱动、.NET 驱动、Ruby 驱动、PostGIS、phpPgAdmin 管理工具和 pgAgent 任务调度器等。

EnterpriseDB 提供了两个版本的 PostgreSQL 安装包：一个是官方开源版，也叫社区版；另一个是商业版，也叫 Advanced Plus 版。后者提供了对 Oracle 语法的兼容以及一些比社区版更强大的管理功能。这两个版本是有区别的，下载时请勿弄错。本书中讨论的所有内容都是针对官方开源版，而非闭源的 Postgres Plus Advanced Server 版。不过由于二者出自同源，所以本书绝大部分内容对于后者也是适用的。

BigSQL 是一个开源的 PostgreSQL 发行版，其开发工作主要由 OpenSCG 公司赞助。BigSQL 发行版与 EnterpriseDB 公司的发行版类似，二者都提供 Windows、Linux 和 Mac 下的 64 位安装包。

BigSQL 发行版的历史没有 EnterpriseDB 发行版那么悠久，其目标是致力于提升 PostgreSQL 的互操作性、DevOps 能力以及大数据处理能力。因此，我们可以在该发行版中看到一些其他发行版一般不会原生提供的扩展包，比如 pgTSQL，这是一个与 Microsoft SQL Server 的 T-SQL 语法相同的过程式语言扩展；另外还含有一些用于性能测评和系统监控的扩展包，比如 pgBadger。

该发行版还自带了功能增强的扩展包，比如 PostGIS（包括 ogr_fdw）和 hadoop_fdw、cassandra_fdw、oracle_fdw 等扩展包，另外还有很多过程式语言扩展包。

同 EnterpriseDB 一样，BigSQL 也有它自己的包管理器。该包管理器可以通过 Web 页面调用，也可以通过一个名为 pgc 的命令行工具调用，pgc 的含义是"非常好的命令行工具"（pretty good command-line）。pgc 包管理工具的用法与 Linux 下的 yum、apt-get 等包管理工具相同，即使在 Windows 环境下用法也是一样。因此如果要安装新的包，请先打开一个新的命令行窗口，然后把目录切换到 BigSQL 的安装目录。

更新并查看本地包列表：

```
pgc update
pgc list
```

输出列表如下：

```
Category     | Component           | Version  | ReleaseDt  | Status    | Cur?
PostgreSQL     pg92                  9.2.21-1   2017-05-11               1
PostgreSQL     pg93                  9.3.17-1   2017-05-11               1
PostgreSQL     pg94                  9.4.12-1   2017-05-11               1
PostgreSQL     pg95                  9.5.7-1    2017-05-11               1
PostgreSQL     pg96                  9.6.3-1    2017-05-11   Installed   1
Extensions     cassandra_fdw3-pg96   3.0.1-1    2016-11-08               1
Extensions     hadoop_fdw2-pg96      2.5.0-1    2016-09-01               1
Extensions     oracle_fdw1-pg96      1.5.0-1    2016-09-01               1
Extensions     orafce3-pg96          3.3.1-1    2016-09-23               1
Extensions     pgaudit11-pg96        1.1.0-2    2017-05-18               1
Extensions     pgpartman2-pg96       2.6.4-1    2017-04-15               1
Extensions     pldebugger96-pg96     9.6.0-1    2016-12-28               1
Extensions     plprofiler3-pg96      3.2-1      2017-04-15               1
Extensions     postgis23-pg96        2.3.2-3    2017-05-18   Installed   1
Extensions     setuser1-pg96         1.2.0-1    2017-02-23               1
Extensions     tds_fdw1-pg96         1.0.8-1    2016-11-23               1
Servers        pgdevops              1.4-1      2017-05-18   Installed   1
Applications   backrest              1.18       2017-05-18               1
Applications   ora2pg                18.1       2017-03-23               1
Applications   pgadmin3              1.23.0a    2016-10-20   Installed   1
Applications   pgagent               3.4.1-1    2017-02-23               1
Applications   pgbadger              9.1        2017-02-09               1
Frameworks     java8                 8u121      2017-02-09               1
Frameworks     perl5                 5.20.3.3   2016-03-14               1
Frameworks     python2               2.7.12-1   2016-10-20   Installed   0
Frameworks     tcl86                 8.6.4-1    2016-03-11               1
```

安装一个包：

```
pgc install pgdevops
```

pgDevops 包是一个基于 Web 的管理工具，其中包含了 pgAdmin4 以及用于安装和监控 BigSQL 相关包的管理界面。

安装好以后，请执行：

```
pgc init pgdevops
pgc start pgdevops
```

装好以后默认的访问路径是：http://localhost:8051。

如需升级一个现有的包，请使用 `pgc upgrade` 代替 `pgc install`。



如果希望在同一台机器上免安装试用一下不同版本的 PostgreSQL，或者是希望从 USB 设备启动 PostgreSQL，EnterpriseDB 和 BigSQL 均提供了一种免安装的解决方案。EnterpriseDB 方案的具体内容请参考“Starting PostgreSQL in Windows without Install”这篇文章。

A.2 CentOS、Fedora、Red Hat以及Scientific Linux

大多数 Linux/Unix 发行版会在其软件仓库中提供 PostgreSQL，但版本可能比较老。为了解决这个问题，很多人会使用逆向移植[1]的版本包，这种版本包的软件仓库中会提供较新版本的 PostgreSQL 软件。

对于具有冒险精神的 Linux 用户来说，可以从 PostgreSQL Yum 仓库下载最新版本的 PostgreSQL，包括开发中的版本。此仓库中不仅包含 PostgreSQL 服务器核心组件，还包含比较常用的扩展包。PostgreSQL 的开发团队负责维护这个仓库，并且会在第一时间发布补丁和版本更新。PostgreSQL Yum 仓库一般会存储最近的 2~4 个稳定版 PostgreSQL，支持的操作系统平台包括 CentOS、RedHat EL、Fedora、Scientific Linux、Amazon AMI 以及 Oracle Enterprise。

如果使用的操作系统平台较老或者是仍需要生命周期已经结束的老版本 PostgreSQL，那么是无法使用最新的 PostgreSQL Yum 仓库的，请查阅文档了解下哪些软件仓库中仍然在维护老版本的 PostgreSQL。如果希望了解更多基于 Yum 的软件安装机制，请参考我们的 PostgresOnline 网站上关于 Yum 的内容。

A.3 Debian和Ubuntu

Debian 和 Ubuntu 上可以使用 apt-postgresql 仓库来安装最新的稳定版或者开发版 PostgreSQL。apt-postgresql 也是个软件仓库，其作用与 PostgreSQL 开发组自己维护的 yum postgresql 软件仓库类似。Ubuntu 和 Debian 自身的默认软件仓库中一般也会提供最新的稳定版 PostgreSQL。典型的安装命令如下：

```
sudo apt-get install postgresql-9.6
```

注 1：“逆向移植”英文为 backport，是指将新版本的软件——比如数据库软件、工具软件、补丁包等——逆向移植到老版本的操作系统上，这样构建出来的版本包既能维持平台兼容性，又能提供新的软件特性，从而以最小的代价解决了老版本软件存在的问题，与全面升级操作系统平台相比，这是一种兼容性好、风险低的解决方案。——译者注

如果你需要编译软件仓库中未提供的扩展包，需要先安装 postgresql-server-dev 开发库，命令如下：

```
sudo apt-get install postgresql-server-dev-9.6
```

如果你的操作系统的默认软件仓库中未包含最新版本的 PostgreSQL，那么请访问 Apt PostgreSQL packages 站点以获取最新的稳定版或者测试版 PostgreSQL。该站点同时还提供一些其他的安装包，比如 PL/V8 和 PostGIS 等。该站点的软件包所支持的操作系统一般包含 Debian 和 Ubuntu 的最近 2~3 个版本。

A.4 FreeBSD

FreeBSD 是一个常用的 PostgreSQL 平台。你可以从 http://www.freebsd.org/ports/database.html 获取到最新的适用于 FreeBSD 平台的 PostgreSQL，然后通过 FreeBSD 的包管理系统来安装。

A.5 macOS

在 Mac 机器上安装 PostgreSQL 有很多方法：EnterpriseDB 和 BigSQL 均提供了独立安装包；Homebrew 包管理器使用得越来越广泛，并且已经吸引了一些高级 Mac 用户；Postgres.app 是 Heroku 封装的一个发行包，在新手用户中很流行；另外历史悠久的 MacPorts 和 Fink 这两个软件发布平台也还在继续提供服务。可以看到，Mac 可用的安装源很多，但我们建议 Mac 用户最好从相同的数据源下载安装包。比如，你使用 BigSQL 提供的安装包安装了 PostgreSQL，那么最好不要用 EnterpriseDB 的 StackBuilder 包管理工具来安装扩展包，因为可能会出现兼容性问题。

以下列出了每个安装源的详细信息。

- EnterpriseDB 公司为 macOS 提供了一个非常易用的一键式 PostgreSQL 安装包，附带了 pgAdmin 管理工具。此外该公司还提供了一个名为 StackBuilder 的软件，通过它可以下载常用扩展包、驱动程序、编程语言扩展以及管理工具等。
- BigSQL 也提供了一个非常易用的一键式安装包，支持 64 位 macOS。如需安装扩展包，BigSQL 提供了一个名为 `pgc` 的命令行工具以及一个名为 pgDevops 的 Web 管理工具，其详情 A.1 节已经讨论过。通过它可以下载常用扩展包、驱动程序、编程语言扩展以及管理工具等。BigSQL 当前在非 Windows 环境中支持 PL/V8 扩展包。
- Homebrew 是 macOS 下的一个安装包管理器，支持包括 PostgreSQL 在内的很多软件的安装管理。“PostgreSQL, Homebrew, and You”这篇博客文章介绍了如何使用 Homebrew 来安装 PostgreSQL。你还能在 Homebrew PostgreSQL Wiki 站点看到大量有价值的相关文章。
- 由 Heroku 开发团队提供的 PostgreSQL.app 是一个免费的桌面应用安装包，号称是 Mac 平台上最方便易用的 PostgreSQL 安装包。该安装包一般都是基于最新版本的 PostgreSQL 构建，其中还附带了常用的扩展包，比如 PostGIS、PL/Python 和 PL/V8 等。Postgres.app 作为一个独立的应用程序运行，可以按需启动和停止，非常适合用于开发，也适用于单用户场景。

- MacPorts 是 macOS 平台上的一个软件发布平台，支持大量的开源软件，可以实现软件包的编译、安装、升级等操作。它是 Mac 操作系统上最早支持 PostgreSQL 的软件发布平台。
- Fink 是 macOS 上的另一个软件发布平台，底层基于 Debian 的 apt-get 软件安装框架。

附录B

PostgreSQL自带的命令行工具

以下内容集中介绍了 PostgreSQL 的必备命令行工具，本书的正文部分已经对它们的功能进行了翔实的介绍，此处仅列出它们的帮助信息。我们希望通过提供这部分内容为你节省一些时间，也希望能让这本书成为你更好的工作助手。

B.1 使用pg_dump进行数据库备份

pg_dump 可备份一个 database 的全部或者部分数据。支持的备份格式有：TAR 包格式、PostgreSQL 自定义压缩格式、纯文本格式以及 SQL 文本格式。纯文本格式转储的内容中含有 psql 专有命令行，因此恢复时也需要通过 psql 工具来执行此文本。SQL 文本格式转储的是仅包含标准 CREATE 和 INSERT 命令的 SQL 脚本，恢复时你可以使用 psql 或者 pgAdmin 工具来运行该脚本。示例 B-1 显示的是 pg_dump 命令的帮助信息。如果要了解 pg_dump 的全部用法，请参见 2.7.1 节。

示例 B-1 pg_dump 帮助信息

```
pg_dump --help

pg_dump可将某个database转储为文本文件或者其他格式文件。
用法:
pg_dump [选项]... [database名]

通用选项:
-f, --file=FILENAME            输出文件名或者目录名
-F, --format=c|d|t|p           输出文件格式（自定义格式、目录格式、TAR包格式、纯文
本）
-j, --jobs=NUM                 使用这多个并行作业进行转储
-v, --verbose                  详细信息模式
-Z, --compress=0-9             压缩格式的压缩级别
```

```
--lock-wait-timeout=TIMEOUT      等待表锁超时后操作失败
--no-sync                        不等待变更安全写入磁盘 ❶
--help                           显示此帮助信息并退出
--version                        输出版本信息并退出
控制输出内容的选项：
-a, --data-only                  仅转储数据，而不转储schema
-b, --blobs                      在转储中包含大对象
-B, --no-blobs                   不对大对象进行备份 ❷
-c, --clean                      在重新创建数据库对象之前清除（删除）数据库对象
-C, --create                     包含用于在转储中创建数据库的命令
-E, --encoding=ENCODING          以ENCODING编码格式转储数据
-n, --schema=SCHEMA              仅转储命名schema
-N, --exclude-schema=SCHEMA      不转储命名schema
-o, --oids                       在转储中包含OID
-O, --no-owner                   以纯文本格式跳过对象所有权的恢复
-s, --schema-only                仅转储schema，而不转储数据
-S, --superuser=NAME             要以纯文本格式使用的超级用户名
-t, --table=TABLE                仅转储命名表
-T, --exclude-table=TABLE        不转储命名表
-x, --no-privileges              不转储特权（grant/revoke）
--binary-upgrade                 仅供升级工具使用
--column-inserts                 以带有列名的INSERT命令的形式转储数据
--disable-dollar-quoting         禁用美元（符号）引号，而是使用SQL标准引号
--disable-triggers               在仅恢复数据期间禁用触发器
--enable-row-security            启用行级安全控制（只导出用户有权访问的数据）❸
--exclude-table-data=TABLE       不转储命名表中的数据
--if-exists                      删除对象时使用IF EXISTS
--inserts                        以INSERT命令（而非COPY命令）的形式转储数据
--no-publications                不导出逻辑复制发布端数据源定义 ❹
--no-security-labels             不转储安全标签分配
--no-subscriptions               不导出逻辑复制订阅端的数据订阅定义 ❺
--no-synchronized-snapshots      在并行作业中不使用同步快照
--no-tablespaces                 不转储表空间分配
--no-unlogged-table-data         不转储不记录WAL日志的表的数据
--quote-all-identifiers          所有标识符加引号，即使不是关键字也加
--section=SECTION                转储命名部分（包括三个部分：pre-data、data以及post-
                                 data。data部分包含表记录数据、大对象数据以及序列的
                                 值；post-data部分包含索引、触发器、规则和约束（除了
                                 验证检查约束）的定义；pre-data部分包含此外其他所有
                                 的对象定义）
--serializable-deferrable        等待直至转储正常运行为止
--snapshot=SNAPSHOT              为导出使用指定的快照 ❻
--strict-names                   要求每个表和/或schema包括模式以匹配至少一个实体 ❼
--use-set-session-authorization  使用 SESSION AUTHORIZATION 命令代替ALTER OWNER
                                 命令来设置所有权
连接选项：
-d, --dbname=DBNAME              要转储的数据库
-h, --host=主机名                数据库服务器主机或套接字目录
-p, --port=端口号                数据库服务器端口号
-U, --username=名称              作为指定数据库用户连接
-w, --no-password                永远不提示输入密码
-W, --password                   强制要求输入密码（应该自动发生）
--role=ROLENAME                  在转储之前执行SET ROLE命令
```

❶❷❸ PostgreSQL 10 中引入的新特性。

❹ PostgreSQL 9.6 中引入的新特性。
❺❻ PostgreSQL 9.5 中引入的新特性。
❼ PostgreSQL 9.4 中引入的新特性。

B.2 服务器级备份工具pg_dumpall

使用 `pg_dumpall` 工具可以将服务器上的所有数据库备份到单个纯文本文件或者单个纯文本 SQL 文件上。该备份工具将自动备份角色和表空间等系统级对象的信息，这类信息不属于任何一个数据库。示例 B-2 列出了 `pg_dumpall` 的所有帮助信息。`pg_dumpall` 的具体用法请参考 2.7.2 节。

示例 B-2　pg_dumpall 帮助信息

```
pg_dumpall --help

pg_dumpall可以将一个PostgreSQL数据库集群中的所有数据都提取到一个SQL脚本文件中。 用
法:
pg_dumpall [选项]...

通用选项:
-f, --file=FILENAME                  输出文件名
-v, --verbose                        详细模式
-V, --version                        输出版本信息，然后退出
--lock-wait-timeout=TIMEOUT          等待表锁超时后操作失败
-?, --help                           显示此帮助信息并退出

控制输出内容的选项:
-a, --data-only                      仅转储数据，而不转储schema
-c, --clean                          重新创建数据库之前清除（删除）数据库
-g, --globals-only                   仅转储全局对象，而不转储数据库
-o, --oids                           在转储中包含OID
-O, --no-owner                       以纯文本格式跳过对象所有权的恢复
-r, --roles-only                     仅转储角色，而不转储数据库和表空间
-s, --schema-only                    仅转储schema，而不转储数据
-S, --superuser=NAME                 要在转储中使用的超级用户名
-t, --tablespaces-only               仅转储表空间，而不转储数据库和角色
-x, --no-privileges                  不转储特权（grant/revoke）
--binary-upgrade                     仅供升级工具使用
--column-inserts                     以带有列名的INSERT命令的形式转储数据
--disable-dollar-quoting             禁用美元（符号）引号，而是使用SQL标准引号
--disable-triggers                   在仅恢复数据期间禁用触发器
--inserts                            以INSERT命令（而非COPY命令）的形式转储数据
--no-publications                    不导出逻辑复制发布端数据源定义 ❶
--no-security-labels                 不转储安全标签的分配
--no-subscriptions                   不导出逻辑复制订阅端的数据订阅定义 ❷
--no-sync                            不等待变更安全写入磁盘 ❸
--no-security-labels                 不转储安全标签分配
--no-tablespaces                     不转储表空间分配
--no-unlogged-table-data             不转储不记录WAL日志的表的数据
--no-role-passwords                  不转储角色的密码 ❹
--quote-all-identifiers              所有标识符加引号，即使不是关键字也加
--use-set-session-authorization      使用SET SESSION AUTHORIZATION命令代替ALTER OWNER
                                     命令来设置所有权
```

```
连接选项：
-d, --dbname=CONNSTR              使用连接连接串连接
-h, --host=主机名                 数据库服务器主机或套接字目录
-l, --database=DBNAME             替代默认数据库
-p, --port=端口号                 数据库服务器端口号
-U, --username=名称               作为指定数据库用户连接
-w, --no-password                 永远不提示输入密码
-W, --password                    强制要求输入密码（应该自动发生）
--role=ROLENAME                   在转储之前执行SET ROLE命令

如果未使用-f/--file，则会将SQL脚本写到标准输出中。
```

❶❷❸❹ PostgreSQL 10 中引入的新特性。

B.3 database数据恢复工具pg_restore

可以使用 pg_restore 可恢复使用 pg_dump 创建的备份文件，这些备份文件的格式包括 TAR 包格式、自定义压缩格式以及目录格式等。示例 B-3 是 pg_restore 命令的帮助信息。更多有关使用 pg_restore 的实例，请参见 2.7.3 节。

示例 B-3 pg_restore 帮助信息

```
pg_restore --help

pg_restore可以从pg_dump创建的存档中恢复一个PostgreSQL数据库。 用法：
pg_restore [选项]... [文件名]

通用选项：
-d, --dbname=NAME               连接到数据库名称
-f, --file=文件名               输出文件名
-F, --format=c|d|t              备份文件格式（应该是自动的）
-l, --list                      打印存档的汇总目录
-v, --verbose                   详细信息模式
-V, --version                   输出版本信息并退出
-?, --help                      显示此帮助信息并退出

恢复控制选项：
-a, --data-only                 仅恢复数据，而不恢复schema
-c, --clean                     在重新创建数据库对象之前清除（删除）数据库对象
-C, --create                    创建目标数据库
-e, --exit-on-error             恢复期间发生错误时退出，若不设定则默认为继续恢复
-I, --index=NAME                恢复命名索引
-j, --jobs=NUM                  使用这多个并行作业进行恢复
-L, --use-list=FILENAME         将此文件的目录用于选择输出或对输出进行排序
-n, --schema=NAME               仅恢复此schema中的对象
-N, --exclude-schema=NAME       不恢复该schema中的对象 ❶
-O, --no-owner                  跳过对象所有权的恢复
-P, --function=NAME(args)       恢复命名函数
-s, --schema-only               仅恢复schema，而不恢复数据
-S, --superuser=NAME            用于禁用触发器的超级用户名
-t, --table=NAME                恢复命名表（含表和视图等） ❷
-T, --trigger=NAME              恢复命名触发器
-x, --no-privileges             跳过访问特权（grant/revoke）的恢复
```

```
-1, --single-transaction      作为单个事务恢复
--enable-row-security         启用行安全性 ❸
--disable-triggers            在仅恢复数据期间禁用触发器
--no-data-for-failed-tables   如果表创建失败，则不对其进行数据恢复
--no-publications             不导出逻辑复制发布端数据源定义 ❹
--no-security-labels          不恢复安全标签
--no-subscriptions            不导出逻辑复制订阅端的数据订阅定义 ❺
--no-tablespaces              不恢复表空间分配
--section=SECTION             恢复命名部分（包括三个部分：pre-data、data以及post-
                              data。data部分包含表记录数据、大对象数据以及序列的
                              值；post-data部分包含索引、触发器、规则和约束（除了
                              验证检查约束）的定义；pre-data部分包含此外其他所有
                              的对象定义）

--strict-names                要求每个表和/或schema包括模式以匹配至少一个实体 ❻

--use-set-session-authorization      使用SET SESSION AUTHORIZATION命令代替ALTER
                                     OWNER命令来设置所有权

连接选项：
-h, --host=主机名             数据库服务器主机或套接字目录
-p, --port=端口号             数据库服务器端口号
-U, --username=名称           作为指定数据库用户连接
-w, --no-password             永远不提示输入密码
-W, --password                强制要求输入密码（应该自动发生）
--role=ROLENAME               在恢复之前执行SET ROLE命令
```

❶❷❸ PostgreSQL 10 中引入的新特性

❹❺ PostgreSQL 9.6 中引入的新特性。在 9.6 版之前，-t 选项只用于过滤普通表。在 9.6 版中，它拓展支持了外表、视图、物化视图和序列号生成器。

❻ PostgreSQL 9.5 中引入的新特性

B.4 交互模式下的psql命令

示例 B-4 中列出了 psql 在交互模式下支持的命令。请参考 3.1 节和 3.2 节的例子了解其使用方法。

示例 B-4　psql 交互模式下支持的命令

```
\?

通用命令
\copyright               显示PostgreSQL使用和分发条款
\errverbose              以最冗长的形式显示最近的错误消息 ❶
\g [文件] or ;           执行查询（并将结果发送给文件或|管道）
\gexec                   执行策略，然后执行其结果中的每个值 ❷
\gset [PREFIX]           执行查询并将结果存储到psql变量中
\h [名称]                关于SQL命令语法的帮助，*代表所有命令
\gx [文件]               作用与\g相同，但强行要求使用展开模式显示结果 ❸
\q                       退出psql
\crosstabview [COLUMNS]  执行查询并且以交叉表显示结果 ❹
\watch [SEC]             每隔SEC秒执行一次查询
```

```
帮助
  \? [commands]          显示反斜线命令的帮助
  \? options             显示 psql 命令行选项的帮助
  \? variables           显示特殊变量的帮助
  \h [名称]              SQL命令语法上的说明，用*显示全部命令的语法说明

查询缓冲区相关命令
\e [FILE] [LINE]         使用外部编辑器编辑查询缓冲区（或文件）
\ef [FUNCNAME [LINE]]    使用外部编辑器编辑函数定义
\ev [VIEWNAME [LINE]]    用外部编辑器编辑视图定义 ❺
\p                       显示查询缓冲区的内容
\r                       重置（清除）查询缓冲区
\w 文件                  将查询缓冲区写入到文件

输入/输出相关命令
\copy ...                执行SQL COPY，将数据流发送到客户端主机
\echo [字符串]           将字符串写到标准输出
\i 文件                  从文件执行命令
\ir FILE                 与\i类似，但是在脚本中执行时，认为目标文件的位置是当前
                         脚本所在的目录
\o [文件]                将所有查询结果发送到文件或|管道
\qecho [字符串]          将字符串写入到查询输出流，该命令等效于\echo，区别是所有
                         输出将写入由\o设置的输出通道

条件命令 ❻
  \if EXPR               开启一个条件判定块
  \elif EXPR             当前条件判定块中的分支条件判定
  \else                  当前条件判定块中的最终条件判定
  \endif                 条件块结束符

信息查询命令
（选项：S = 显示系统对象，+ =附加的详细信息）
\d[S+]                   输出表、视图和序列列表
\d[S+]  名称             描述表、视图、序列或索引
\da[S]  [模式]           输出聚合函数列表
\dA[+]  [模式]           列出访问方法  ❼
\db[+]  [模式]           输出表空间列表
\dc[S]  [模式]           输出编码转换（conversion）列表
\dC     [模式]           输出类型强制转换（cast）列表
\dd[S]  [模式]           显示对象上的注释
\ddp    [模式]           输出默认权限列表
\dD[S]  [模式]           输出域列表
\det[+] [模式]           输出外部表列表
\des[+] [模式]           输出外部服务器列表
\deu[+] [模式]           输出用户映射列表
\dew[+] [模式]           输出外部数据封装器列表
\df[antw][S+] [模式]     输出特定类型函数（仅a-聚合函数/n-常规函数/t-触发器函数
                         /w-窗口函数）列表
\dF[+]  [模式]           输出文本搜索配置列表
\dFd[+] [模式]           输出文本搜索字典列表
\dFp[+] [模式]           输出文本搜索解析器列表
\dFt[+] [模式]           输出文本搜索模版列表
\dg[S+] [模式]           输出角色列表
\di[S+] [模式]           输出索引列表
\dl                      输出大对象列表，与\lo_list相同
```

```
\dL[S+] [模式]              输出过程式语言列表
\dm[S+] [模式]              输出物化视图列表
\dn[S+] [模式]              输出schema列表
\do[S]  [模式]              输出运算符列表
\dO[S+] [模式]              输出排序规则列表
\dp     [模式]              输出表、视图和序列访问权限列表
\drds [模式1 [模式2]]       输出每个database的角色设置列表
\dRp[+] [PATTERN]           列出逻辑复制的数据源定义 ❽
\dRs[+] [PATTERN]           列出逻辑复制的订阅定义 ❾
\ds[S+] [模式]              输出序列列表
\dt[S+] [模式]              输出表列表
\dT[S+] [模式]              输出数据类型列表
\du[S+] [模式]              输出角色列表
\dv[S+] [模式]              输出视图列表
\dE[S+] [模式]              输出外部表列表
\dx[+]  [模式]              输出扩展列表
\dy     [模式]              输出事件触发器列表
\l[+]                       输出数据库列表
\sf[+] FUNCNAME             显示函数定义
\sv[+]  VIEWNAME            显示一个视图的定义 ❿
\z      [模式]              和\dp的功能相同

格式化相关命令
\a                          在非对齐输出模式和对齐输出模式之间切换
\C [字符串]                 设置表标题；或者如果没有，则不设置
\f [字符串]                 显示或设置非对齐查询输出的字段分隔符
\H                          切换HTML输出模式（当前关闭）
\pset NAME [VALUE]          设置表输出选项
                            （NAME的可选项有format、border、expanded、fieldsep、
                            fieldsep_zero、footer、null、numericlocale、
                            recordsep、tuples_only、title、tableattr、pager、
                            pager_min_lines、recordsep、recordsep_zero、tableattr、
                            title、tuples_only、unicode_border_linestyle、
                            unicode_column_linestyle、unicode_header_linestyle）⓫
\t [on|off]                 仅显示行（当前关闭）
\T [字符串]                 设置HTML
\x [on|off]                 切换扩展输出（当前关闭）

连接相关命令
\c[onnect] {[DBNAME|- USER|- HOST|- PORT|-] | conninfo}
                            连接到新数据库（当前是"postgres"）
\encoding [编码名称]        显示或设置客户端编码
\password [USERNAME]        安全地为用户更改密码
\conninfo                   显示当前连接的相关信息

操作系统相关命令
\cd [目录]                  更改当前工作目录
\setenv NAME [VALUE]        设置或取消设置环境变量
\timing [on|off]            切换命令计时开关（当前关闭）
! [命令]                    在shell中执行命令或启动交互式shell
```

❶❷❸ PostgreSQL 10 中引入的新特性。所有的条件参数都是新引入的。

❹❺❻❼❽❾❿ PostgreSQL 9.6 中引入的新特性。

⓫ PostgreSQL 9.5 中引入的新特性。

B.5 非交互模式下的psql命令

示例 B-5 是非交互模式下 psql 的命令行帮助信息。关于该模式下的具体使用例子请参考 3.2 节。

示例 B-5 psql 基本帮助信息

```
psql --help

psql是PostgreSQL的交互式终端。
使用方法：
psql [选项]... [database名称 [用户名]]

通用选项：
-c,--command=命令                        仅运行单个命令（SQL或内部命令），然后退出
-d, --dbname=数据库名称                  要连接到的数据库名称
-f, --file=文件名                        从文件执行命令，然后退出
-l, --list                               列出可用的数据库，然后退出
-v, --set=, --variable=NAME=VALUE
                                         设置psql变量NAME为VALUE
                                         (例如，-v ON_ERROR_STOP=1)
-X, --no-psqlrc                          不读取启动文件（~/.psqlrc）
-1 ("one"), --single-transaction         将命令文件作为单一事务执行
-?, --help[=options]                     显示此帮助，然后退出
    --help=commands                      列出反斜线命令，然后退出 ❶
    --help=variables                     列出特殊变量，然后退出 ❷
--version                                输出版本信息并退出

输入和输出选项：
-a, --echo-all                           回显所有来自于脚本的输入
-b, --echo-errors                        回显失败的命令 ❸
-e, --echo-queries                       回显发送给服务器的命令
-E, --echo-hidden                        显示内部命令生成的查询
-L, --log-file=文件名                    将会话日志发送给文件
-n, --no-readline                        禁用增强命令行编辑功能（readline）
-o, --output=FILENAME                    将查询结果发送给文件（或|管道）
-q, --quiet                              以静默模式运行（不显示消息，仅显示查询输出）
-s, --single-step                        单步模式（每个查询均需确认）
-S, --single-line                        单行模式（SQL命令不允许跨行）

输出格式选项：
-A, --no-align                           非对齐表输出模式
-F, --field-separator=字符串设置字段分隔符（默认为“|”）
-H, --html                               HTML表输出模式
-P, --pset=VAR[=ARG]                     将打印选项VAR设置为AR（参见\pset命令）
-R, --record-separator=字符串设置记录分隔符（默认为换行符）
-t, --tuples-only                        仅打印行
-T, --table-attr=文本                    设置HTML表标记属性（例如：宽度、边框等）
-x, --expanded                           打开扩展表输出
-z, --field-separator-zero               将字段分隔符设置为零字节
-0, --record-separator-zero              将记录分隔符设置为零字节
```

```
连接选项：
-h, --host=主机名                      数据库服务器主机或套接字目录
-p, --port=端口                        数据库服务器端口（默认为“5432”）
-U, --username=用户名                  数据库用户名
-w, --no-password                      永远不提示输入密码
-W, --password                         强制要求输入密码（应该自动发生）

如需了解更多信息，请在psql中输入“\?”（用于内部命令）或者“\help”（用于SQL命
令），或者参考PostgreSQL官方手册中的psql部分。
```

❶❷❸ PostgreSQL 9.5 中引入的新特性。

作者简介

Regina Obe（瑞金娜·奥贝）是位于美国波士顿的数据库咨询服务公司 Paragon Corporation 的负责人之一。她具有 20 余年的数据库领域从业经验，精通多种编程语言和数据库系统，特别是在空间数据库方面尤为专长。她是 PostGIS 指导委员会成员，同时也是 PostGIS、pgRouting 和 GEOS 核心开发团队的成员。 Regina 拥有麻省理工学院机械工程学士学位，是 *PostGIS in Action* 和 *pgRouting: A Practical Guide* 这两本书的作者之一。

Leo Hsu（利奥·徐）也是 Paragon Corporation 公司的负责人之一。他有着 20 余年的数据库领域从业经验，曾为许多不同规模的公司和组织做过数据库开发工作，对数据库领域有着非常深入的思考和研究。Leo 拥有斯坦福大学经济系统工程硕士学位以及麻省理工学院机械工程与经济学硕士学位，他也是 *PostGIS in Action* 和 *pgRouting: A Practical Guide* 这两本书的作者之一。

封面介绍

本书封面上的动物是象鼩（拉丁名为 Macroscelides proboscideus），这是一种原产于非洲的食虫性哺乳动物，广泛分布于非洲南部，因有着类似大象的长鼻而得名。它们能够适应各种各样的生存环境：无论是纳米布沙漠，还是砾石覆盖的南部非洲地区，甚至茂密的森林地带，都是它们的栖居之地。

象鼩是一种体型很小的四足动物。由于尾巴非常相似，象鼩外表上看起来像是老鼠或者负鼠。相较于其体型来说，它们的腿可以说相当之长，因此它们可以跳跃行走，看起来和兔子很相似。它们的鼻子根据亚种的不同而长度各异，但在寻找食物时都可以左右扭动。

虽然象鼩是一种活跃的昼行性动物，但由于其个性机警，所以一般很难见到或者捕捉到它们。它们很善于伪装，在遇到危险时会迅速逃避。

象鼩并非高度群居性的动物，很多个体都是以一夫一妻的方式结伴生活并共同保护它们的领地。雌性象鼩有着类似人类女性的月经周期，其发情期会持续好几天。雌性个体怀孕以后，其妊娠期会持续 45 至 60 天，一年会生育若干胎，每胎约有 1 到 3 只幼鼩。幼鼩出生时其身体已经发育得比较完全，几天后就会离开巢穴。

出生 5 天之后，幼鼩开始进食昆虫，这些昆虫由它们的母亲捕获并衔在口中携带回来。幼鼩会在出生之后大约 15 天开始尝试独立生活并逐步减少对母亲的依赖。随后它们会圈定自己的领地，并在 41 至 46 天内达到性成熟状态。

成年象鼩主要以无脊椎动物为食，比如昆虫、蜘蛛、蜈蚣、千足虫以及蚯蚓等。要想吃掉个头更大些的猎物对它们来说会有点困难，它们必须用脚拖住猎物，再用牙齿把食物撕扯成碎片，等这些碎片落到地上之后，象鼩会像食蚁兽那样用舌头将它们舔进嘴里。象鼩同时也是植食性动物，如果能找到的话，嫩叶、种子、小型果实等也都是它们的美食。

很多出现在 O'Reilly 图书封面上的动物都濒临灭绝，它们的存在对于维持地球的物种多样性非常重要，如果你希望为保护它们尽一份力量，请访问 animals.oreilly.com 以了解详情。

封面图片来自于 *Meyers Kleines* 词典。

O'REILLY®

PostgreSQL即学即用（第3版）

本书是实用的PostgreSQL快速上手指南，简明扼要地介绍了PostgreSQL核心概念与功能特性，可以帮助读者快速学习、理解并运用好这款开源数据库。

除了用诸多示例重点演示如何实现在别的数据库中难以实现或者根本不可能实现的任务，作者还介绍了ANSI-SQL语法、外部数据封装器增强、全文检索函数和运算符、XML语法、逻辑复制等PostgreSQL 9.5之后版本的新增功能，充分展示了PostgreSQL作为全栈数据库的强大功能和企业级特性。

- **执行基本的数据库管理任务，比如角色管理、数据库创建、数据备份和恢复等；**
- **使用psql命令行工具以及pgAdmin图形化管理工具；**
- **PostgreSQL的表、约束和索引等数据库对象的特性和使用方法；**
- **PostgreSQL所特有的若干功能强大的SQL语法；**
- **使用多种不同的编程语言来编写PostgreSQL函数；**
- **实施语句调优以充分挖掘服务器硬件的潜能；**
- **通过外部数据封装器来查询多种多样的外部数据；**
- **使用系统内置的复制筛选器复制数据。**

瑞金娜·奥贝（Regina Obe）是数据库咨询公司Paragon的负责人之一，在多种编程语言和数据库系统上有20多年的从业经验。她是PostGIS项目指导委员会成员，也是PostGIS核心开发团队的成员，还参与pgRouting和GEOS开发。

利奥·徐（Leo Hsu）也是Paragon公司的负责人之一，曾为大大小小的组织开发过数据库，有20多年的从业经验。

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　**计算机／数据库**

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版





ISBN 978-7-115-49966-0
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks